Aptech Certified
Computer Professional

Object Oriented Programming with C++

Scanned by CamScanner

005.532 A533

Student Guide

9905
Semester – II
Multi – Modal
Object Oriented Programming with C++

Name :- M. Yusuf M. Baluch.

Batch :- B.S.I.T.

Semester :- II

I.D. :- CF/200002A0004.

Durr-e-Danish Library

Scanned by CamScanner

© APTECH WORLDWIDE
All rights reserved. No part of this book may be reproduced in any manner whatsoever, stored in a retrieval system or transmitted or translated in any form or manner or by any means, without the prior written permission of
APTECH WORLDWIDE

APTECH WORLDWIDE
2nd Floor, Najeeb Centre (Above McDonalds)
P.E.C.H.S., Tariq Road, Karachi. PAKISTAN

PRINTED IN PAKISTAN
1st Edition - July 2000

Durr-e-Danish Library

Scanned by CamScanner

Introduction to the Module

Object-oriented programming has brought about a great change in the way software is developed. Large programs are the bane of a programmer's existence, especially if they have to be revised or debugged by someone other than the original developer. Object-oriented programming promises to make complex systems easier to structure and manage. The goal of object-oriented programming is to make programs clearer to understand, more reliable and easier to maintain. Object-oriented concepts give you a new insight to software development because they help you to look at applications from the real-life point of view instead of just in terms of data types and functions.

One of the most popular object-oriented programming languages is C++. C++ was developed by Bjarne Stroustrup using C as a base but implementing the principles of object-orientation most effectively to give you a powerful language. Although C was used as the base for developing C++, you will be surprised to see that there is very little overlap between the two languages' features. While teaching the concepts of C++, we have endeavoured to relate the details of the language to object-oriented concepts. This will help you to understand why certain features of the language exist.

Durr-e-Danish Library

Scanned by CamScanner

Session Objectives:

At the end of this session, the student will be able to -

- *Discuss the following*
  - *The Object-Oriented approach*
  - *Drawbacks of traditional programming*
  - *Object-Oriented programming*
- *Discuss basic Object-Oriented concepts such as:*
  - *Objects*
  - *Classes*
    - *Properties*
    - *Methods*
  - *Abstraction*
  - *Inheritance*
  - *Encapsulation*
  - *Polymorphism*
- *Compare Classes with Structures*
- *Describe Private and Public sections of Classes*
- *Define Member functions*
- *Use the Objects and Member functions of a Class*
  - *Define Objects*
  - *Access Member Functions*
  - *Pass and return Objects*
- *Discuss briefly the features of C++ and another OO language (Smalltalk)*

1.1 Introduction

1.1.1 The Object-Oriented approach

All around us in the real world are objects. We can classify living beings as objects just as we classify things we use as objects of different types. Let us think of departments as objects in an organisation. Typically an organisation has departments that deal with administration, sales, accounts, marketing and so on. Each department has its own personnel who perform clearly assigned duties. Each department also has its own data such as personnel records, inventory, sales figures, or other data related to the functioning of that department.

Durr-e-Danish Library

Scanned by CamScanner

When the whole organisation is split up into departments it is easier to manage activities and personnel. The people in each department control and operate on that department's data. The accounts department is in charge of salaries for the organisation. If you are from the marketing division and need to find out details regarding the salaries of your division all you need to do is enquire in the accounts department for the relevant information. This way you are ensured that some qualified person from the accounts department accesses the data and provides you with the information. It is also a good idea that no unauthorised person from another department accesses the data or tries to make any changes that might corrupt the data.

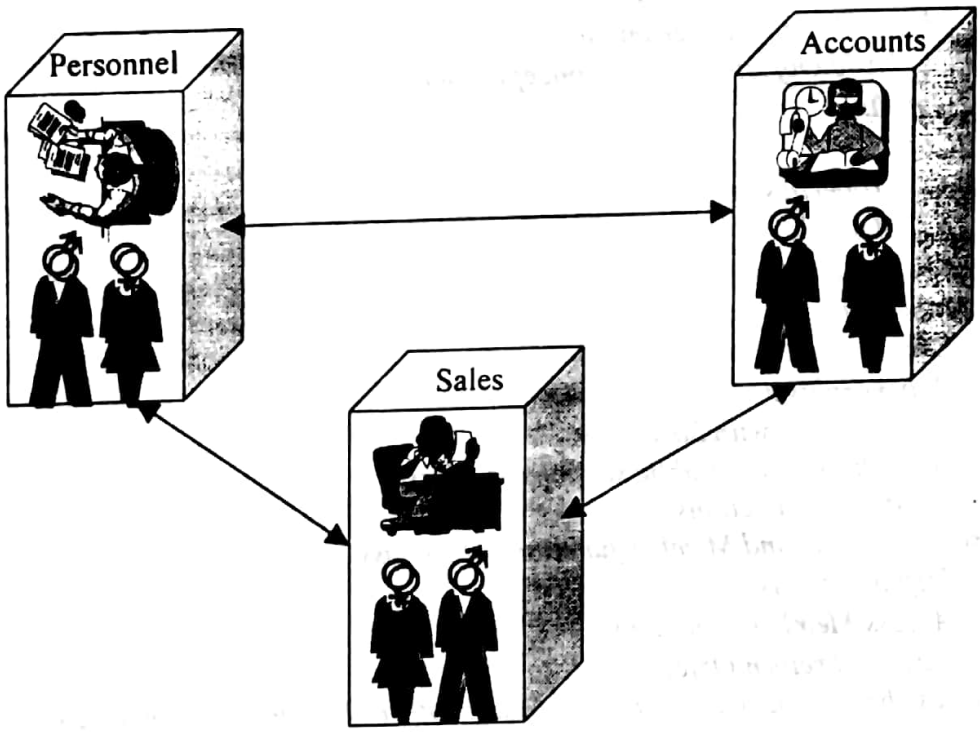


Figure 1: A typical organisation

This concept of objects can be extended to almost every walk of life and more so to program organisation. Any application can be defined in terms of entities or objects so that the process replicates human thought process as closely as possible.

Let us first see how programming is done traditionally and then move on to the changes that object-orientation can bring about in programming.

1.1.2 Drawbacks of Traditional programming

- **Unmanageable programs**

Traditional programs consist of a list of instructions written in a language that tells the computer to perform some actions. When programs become larger they become unmanageable. For this reason functions or procedures or subroutines were adopted to make programs more comprehensible to programmers. A program is divided into functions with clearly defined purposes. Each function can interact with another function or act upon given data. As programs grow ever larger and more complex, even the use of functions can create problems.

Durr-e-Danish Library
Scanned by CamScanner

In the case of an organisation each department performs different tasks like:

- Recruitment of personnel
- Maintaining personnel records
- Paying salaries
- Purchase of materials
- Sale of manufactured goods
- Checking inventory

The list can vary depending on the nature of work done in an organisation. Typically each distinct task can be represented as a function in an application.

- **Problems in modification of data**

Each of the tasks in an organisation act on some information in the form of data. Data plays a crucial role in traditional programming techniques. As we have seen in the case of an organisation the data available in each department is important.

Some or all functions of one or more departments may need to access certain common data. If you add new data items, you will need to modify all the tasks or functions that access the data so that they can also access these new items. For example, personnel records are usually the kind of data that several departments will need to access at sometime or the other. As a result, when personnel records are revised there are a number of functions that will be affected.

Similarly, in a programming application, when data is made available to all functions, it becomes difficult to locate all such functions, and even harder to modify them correctly as the size of the program increases. This is especially difficult when a new programmer tries to understand the intricacies of a large program.

What is needed is a way to restrict access to the data so that only a few critical functions act upon it. This is similar to the analogy we have seen in which salaries are the under the total control of only the accounts department.

The requirements of a system are always being revised. When there are revisions in a program it should not affect the whole system. For example, if the sales department were to make changes in their sales strategy, the change should in no way affect the way salaries are paid to the personnel. In traditional programming techniques it is next to impossible to separate parts of a program from revisions made in another part.

- **Difficulty in implementation**

Traditionally, code and data have been kept apart. For example, in the C language, units of code are called *functions*, while units of data are called *structures*. Functions and structures are not formally connected in C.

The focus of the traditional programming approach tends to be on the implementation details. However, functions and data structures do not model the real world adequately. Humans are not used to thinking in terms of functions and data. The focus of a person's thinking is usually in terms of things or entities, their properties and actions. When we think of an application involving, say, the accounts department of an organisation we think in terms of:

Durr-e-Danish Library

Scanned by CamScanner

- The cashier pays the salaries
- An employee submits vouchers
- The accounts officer sanctions payments
- The cashier tallies accounts

It is difficult to decide how to implement this in terms of data structures, variables and functions. If there were a method by which functions and data at the programming level could correspond to real-world entities and actions it would simplify the problem. This is where object-oriented programming techniques help.

1.1.3 Introduction to Object -Oriented Programming

Object-oriented programming is a reaction to programming problems that were first seen in large programs being developed in the 70s. It offers a powerful model for writing computer software. Object-oriented programming allows for the analysis and design of an application in terms of entities or objects so that the process replicates the human thought process as closely as possible.

This means that the application has to implement the entities as they are seen in real life and associate actions and attributes with each. In object-oriented programming, code and data are merged into a single indivisible thing -- an object. When you approach a programming problem in an object-oriented language, you no longer ask how the problem will be divided into functions or procedures, but how it will be divided into objects.

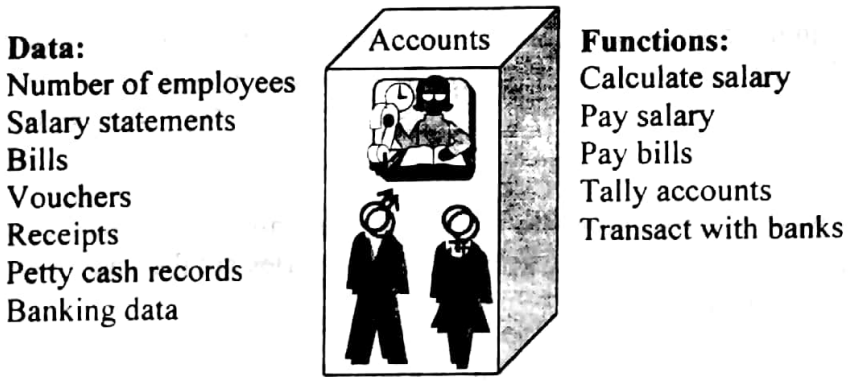


Figure 2: Data and functions of an object

As we look at various concepts related to object-orientation you will see that there is a close match between objects in the programming sense and objects in the real world.

1.2 Basic Object-Oriented concepts

There are several concepts underlying object-oriented technology. These concepts have a one-to-one correspondence between the way we think about a problem and the implementation.

1.2.1 Objects

An object represents an entity in the real world. An object is simply something that makes sense in a particular application.

Durr-e-Danish Library
Scanned by CamScanner

Definition

An *object* is a concept or thing with defined boundaries that is relevant to the problem we are dealing with.

Objects serve two purposes:

1. They help to understand the real world
2. They provide a practical basis for a computer application

To see how real-life entities or things can become objects in object-oriented programs let us look at some typical examples listed below.

- Physical objects
  - Vehicles in a traffic-monitoring application
  - Electrical components in a circuit design problem
  - Countries in a global weather model

- Elements of the computer-user environment
  - Windows
  - Menus
  - Graphics objects
  - The mouse and the keyboard

- Collections of data
  - An inventory of machine parts
  - A personnel file
  - A table of marks relating to an examination

- User-defined data types
  - Time
  - Angles
  - Complex numbers

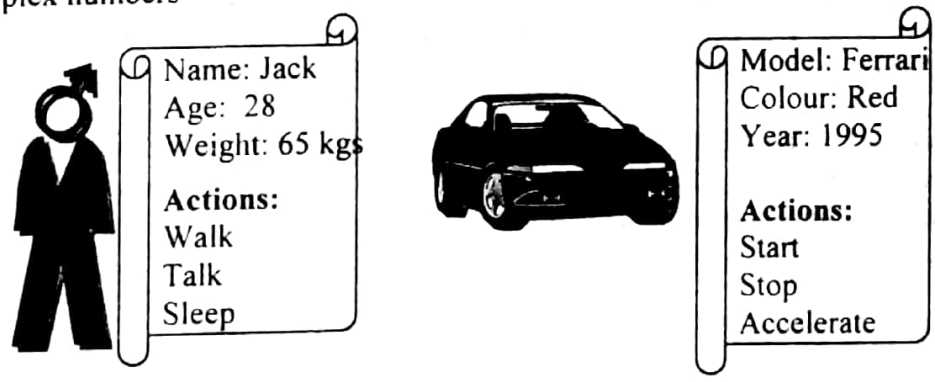


Figure 3: A "Person" object and a "Car" object

Each object has its own properties or characteristics that describe what it is or does. For example the properties of a Person object would be:

- Name

Durr-e-Danish Library

Scanned by CamScanner

- Age
- Weight

Some properties of a Car object would be:

- Colour
- Weight
- Model-year
- Number of wheels
- Engine power

An object also executes some actions. The actions a Car is capable of are:

- Start
- Stop
- Accelerate
- Reverse

The match between programming objects and real-world objects is the result of combining the properties and actions of an object.

1.2.2 Classes

Objects with similar properties and actions need to be grouped together into a unit that can be used in a program. Similar objects or objects with common properties, are grouped into a class. Each class describes a set of individual objects. The term class is an abbreviation of "class of objects". A class of persons, class of animals, class of processes, class of polygons and class of window objects are all examples of classes.

Definition
A *class* is a grouping of objects that have the same properties, common behaviour and common relationships.

The class defines the characteristics that the object is to possess. However, values can be assigned only after an object is created. Only when the object is created does an actual instance of the entity come into existence.

Each object is said to be an *instance* of its class. For example, in a class of Persons, specific individuals, say, Mary, Jack, Joe etc. are the objects. In a class of Polygons, a triangle, a square, a parallelogram, a rectangle and so on, are objects.

Durr-e-Danish Library
Scanned by CamScanner

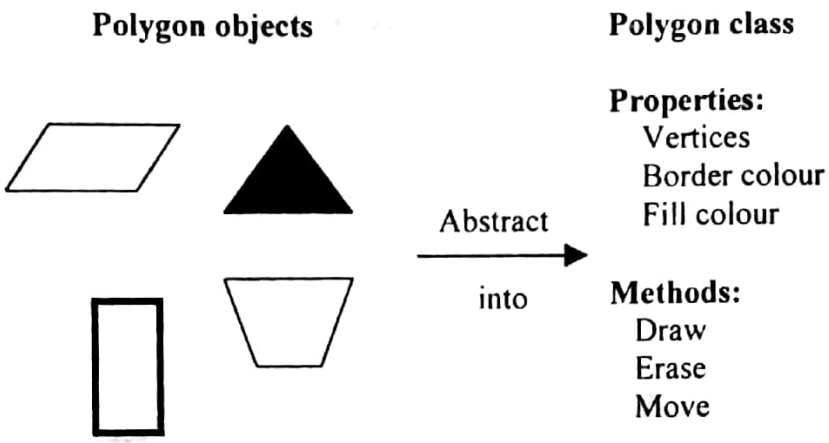


Figure 4: Objects and classes

- **Property/Attribute**

The characteristics of the object are represented as the variables in a class and referred to as the *properties* or *attributes* of the class. For example, each person in a class has a name, an age, IQ and may work at a job. These are the common properties that all persons in the class share. Similarly, in a class of polygons all objects have vertices and edges.

Definition

A characteristic required of an object or entity when represented in a class is called a *property*.

A class is a prototype and not an actual specimen of the entity. Each instance of the class or object has its own value for each of its properties but it shares the property names or operations with other instances of the class.

- **Method**

The actions that are required of an object have also to be programmatically represented in the corresponding class. All objects in a class perform certain common actions or operations. Each action required of an object becomes a function in the class that defines it and is referred to as a *method*. In the polygon class above "draw", "erase" and "move" are examples of the methods that are part of the class.

Definition

An action required of an object or entity when represented in a class is called a ***method***.

In the example of an organisation each department performs some actions like:

- Managing the operations of the department
- Filing data
- Sending out memos

These actions or methods are common for each department apart from specific duties assigned to them.

Durr-e-Danish Library
Scanned by CamScanner

You can think of an object as a "black box" which receives and sends *messages*. The black box actually contains *code* (sequences of computer instructions) and *data* (information which the instructions operates on).

In the example of an organisation, each department is considered an object. Information passed to and retrieved from each department either by inter-departmental memos or verbal instructions are the messages between objects. These messages can be translated to function calls in a program.

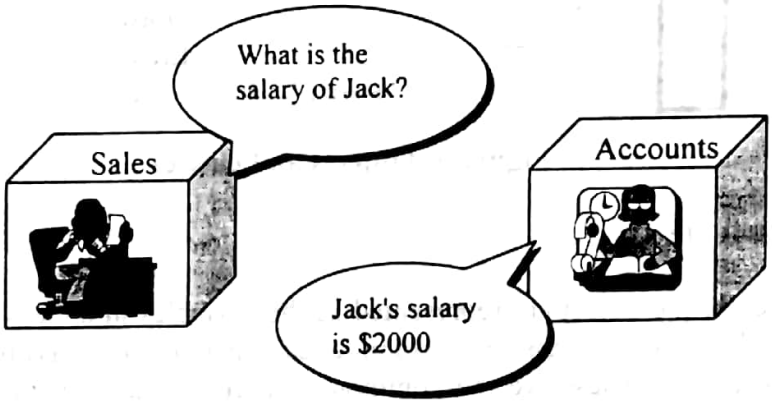


Figure 5: Objects sending messages to each other

It is a fact that the class is such a generalised concept that there are libraries of prewritten classes available in the marketplace. You can purchase classes that perform some generalised operations such as managing stacks, queues, or lists, sorting data, managing windows, etc. This is because of the generality and flexibility of the class construct.

1.2.3 Abstraction

Abstraction is the process of examining certain aspects of a problem. In system development, this means focussing on what an object is and does, before deciding how it should be implemented. An abstract specification tells us what an object does independent of how it works. Object-oriented languages all implement data abstraction in a clean way using classes.

➢ Data Abstraction

Data Abstraction is the process of examining all the available information about an entity to identify information that is relevant to the application. Data abstraction is used to identify properties and methods of each object as relevant to the application at hand.

Definition

Data abstraction is the process of identifying properties and methods related to a particular entity as relevant to the application.

By grouping objects into classes, we are in fact performing data abstraction of a problem. Common definitions are stored once per class rather than once per instance of the class. Methods can be written once for a class, so that all the objects in a class benefit from code reuse.

Durr-e-Danish Library
Scanned by CamScanner

For example, all ellipses share the same methods to:

- Draw them
- Compute their area
- Test for intersection with a line

Polygons would have a separate set of methods. Even special cases, such as circles and squares, can use the general methods. In object-oriented programming data abstraction is defined as a collection of data and methods, which is what an object represents.

1.2.4 Inheritance

The idea of classes leads to the idea of *inheritance*. In our daily lives, we use the concept of classes being divided into subclasses. We know that a class of animals can be divided into mammals, amphibians, insects, reptiles, etc. A class of vehicles is divided into cars, trucks, buses, and motorcycles. A class of shapes can be divided into lines, ellipses, boxes, etc.

Definition

Inheritance is the property that allows the reuse of an existing class to build a new class.

The principle in this sort of division is that each subclass shares common properties with the class from which it is derived. For example, all vehicles in a class may share similar properties of having wheels and a motor. In addition, the subclass may have its own particular characteristics. For example, a bus may have seats for people, while trucks have space for carrying goods.

This new class *inherits* all the behaviour of the original class. The original class is called the *parent class*, or *superclass*, of the new class. Some more terms - a subclass is said to be a *specialisation* of its superclass, and conversely a superclass a *generalisation* of its subclasses.

Definition

The *superclass* is the class from which another class inherits its behaviour.

Definition

The class that inherits the properties and method of another class is called the *subclass*.

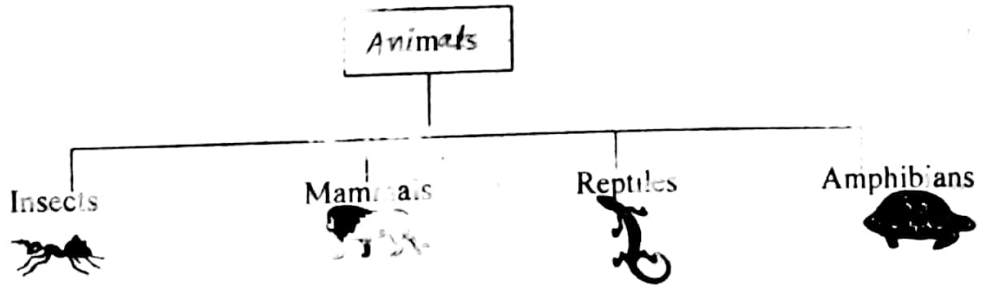


Figure 6: Class Animals and its subclasses

Durr-e-Danish Library
Scanned by CamScanner

1.2.5 Encapsulation

Using the black box as the example of an object, a primary rule of object-oriented programming is this: as the user of an object, you should never need to look inside the box. A class has many properties and methods. It is not necessary for a user to have access to all of them.

All communication to an object is done via messages. The object, which a message is sent to, is called the receiver of the message. Messages define the *interface* to the object. Everything an object can do is represented by its message interface. So you do not have to know anything about what is in the black box in order to use it.

Providing access to an object only through its messages, while keeping the details private is called *information hiding*. An equivalent buzzword is *encapsulation*.

Definition

Encapsulation **is the process that allows selective hiding of properties and methods in a class.**

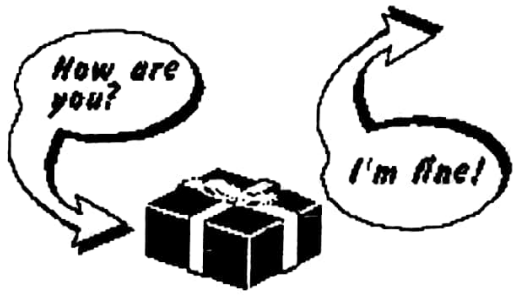


Figure 7: An object's interface via messages

The advantage of encapsulation is that a class can have many properties and methods but only some of these need to be exposed to the user. Thus encapsulation may also be defined as:

Definition

Encapsulation **is the process that allows selective exposing of properties and methods in a class.**

In the example of an organisation there are many methods and properties of the organisation that are classified as private. When another organisation, say, Company B deals with Company A, it does not interfere with the way in which Company A functions. Any information that Company B needs has to be strictly routed through proper channels such as, the public relations officer or administrative manager of Company A.

Looking at it from the programming angle, by not looking inside an object's black box you are not tempted to directly modify that object. If you did, you would be tampering with the details of how the object works. When we properly encapsulate some code we achieve two objectives:

1. We build an impenetrable wall to protect the code from accidental corruption due to the silly little errors that we are all prone to make.
2. We also isolate errors to small sections of code to make them easier to find and fix.

Durr-e-Danish Library
Scanned by CamScanner

Some of the costliest mistakes in computer history have come from software that breaks when someone tries to change it.

1.2.6 Reusability

Along with the abstraction and encapsulation of data and operations comes the aspect of reusability. Object-oriented development allows:

- Information to be shared within an application
- Reuse of designs and code on future projects

All object-oriented languages try to make parts of programs easily reusable and extensible. Programs are broken down into reusable objects. These objects can then be grouped together in different ways to form new programs. Programmers can take an existing class and add additional features to it without modifying the original class. By giving programmers a very clean way to reuse code it is much easier to write new programs by assembling existing pieces.

Inheritance also promotes reuse. You do not have to start from scratch when you write a new program. You can simply reuse an existing range of classes that have behaviours similar to what you need in the new program.

For example, after creating the class Cat, you might make a subclass called Lion, which defines some lion-specific functions, such as "hunt". Or it might make more sense to define a common class called Felines, of which both Cat and Lion are subclasses.

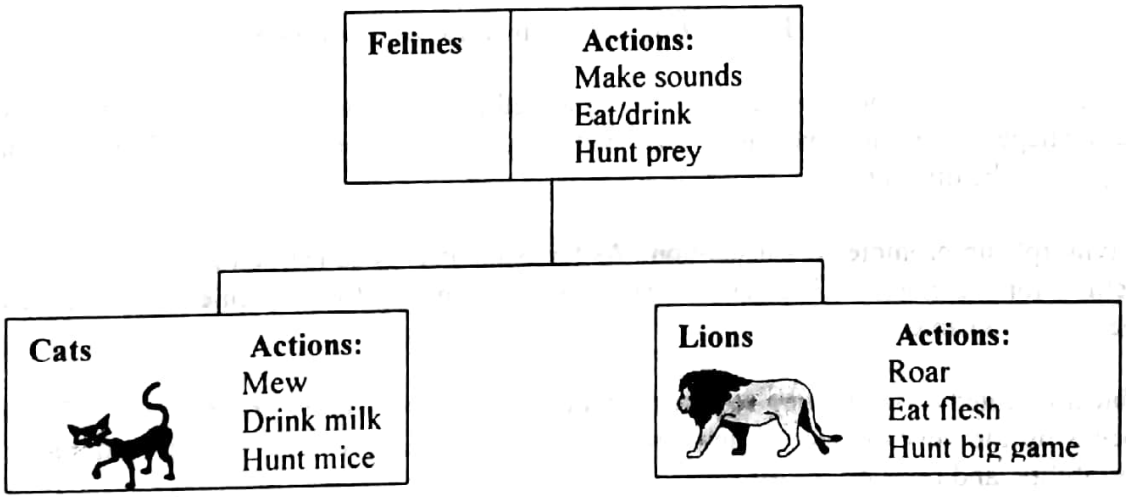


Figure 8: Felines and subclasses

Much of the art of object-oriented programming is in determining the best way to divide a program into an economical set of classes. In addition to speeding up development time, proper class construction and reuse results in far fewer lines of code, which means less bugs and lower maintenance costs.

Durr-e-Danish Library
Scanned by CamScanner

1.2.7 Polymorphism

Object oriented languages try to make existing code easily modifiable without actually changing the code. This is a unique and very powerful concept, because it does not at first seem possible to revise something without changing it. Using *inheritance* and *polymorphism* it is possible to do just that.

Polymorphism means that the same functions may behave differently on different classes. The existing object stays the same, and any changes made are only additions to it. Using this approach a programmer is able to maintain and revise code with less errors since the original object is not changed.

Definition

Polymorphism enables the same function to behave differently on different classes.

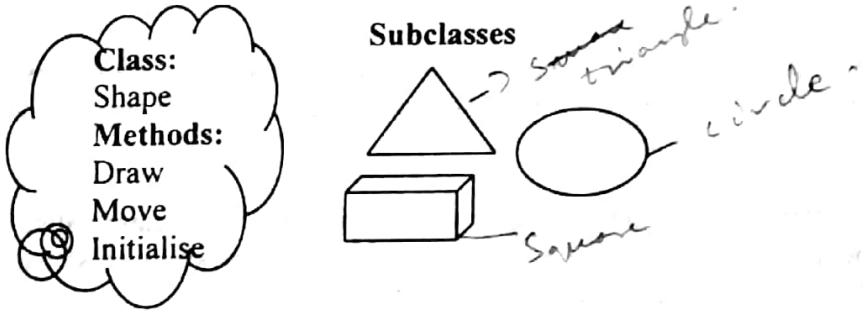


Figure 9: Class Shape and its subclasses

In Figure 9 above, you can see that Draw is a method or function which all the subclasses of the class Shape share. However, the Draw method that is implemented for a box and that for an ellipse will be different.

Polymorphism promotes encapsulation. As far as the user is concerned there needs to be just one method for the class, such as Draw. How the Draw method is implemented for different cases need not be visible to the user.

Object oriented programming requires a major shift in thinking by programmers. This approach speeds up the development of new programs, and, if properly used, improves the maintenance, reusability, and revision of software.

Durr-e-Danish Library
Scanned by CamScanner

1.3 Classes and Structures

The class data type as defined in an object-oriented language such as C++ is very similar to structures. A structure contains one or more data items (called members) which are grouped together as a single unit. On the other hand, a class consists of not only data elements but also functions, which operate on the data elements. The data and functions can be defined in a class as one of the sections such as private and public. The general syntax of the class construct is:

```
class user_defined name{
      private:
            data_type members;
            implementation operations;
      public:
            data_type members;
            implementation operations;
      };
```

The private and public sections of a class are given by the keywords private and public respectively. All the variables and the functions declared in a class, whether in the public or private section, are the members of the class. The keywords private and public in a class determine the accessibility of class members. We will see more of this in a little while.

In a structure all elements are public by default. Consider implementing the concept of a date using a struct to define the representation of a date and a set of functions to manipulate variable of this type. In the example below, there are functions to set the date to a particular value and print the date.

```
struct date {int month, day, year;};
date today;
void set_date(date*, int, int, int);
void print_date(const date*);
//.....
```

The set of functions to manipulate date does not specify that those functions should be the only ones to access objects of type date. This restriction can be expressed by using a class instead of a struct. For example:

```
class date {
      private:
            int month, day, year;
      public:
            void set(int, int, int);
            void print();
      };
```

1.4 Private and Public sections of a class

Class members can be declared in the public or the private sections of a class. As one of the features of object-oriented programming is to prevent data from unrestricted access, the data members of a class are normally described in the private section. Information hiding is the insulation of data members from direct access in a program.

Durr-e-Danish Library

Scanned by CamScanner

The public part constitutes the *interface* to objects of the class. The data members and functions declared in the public section, can be accessed by any function in the outside world (outside of the class).

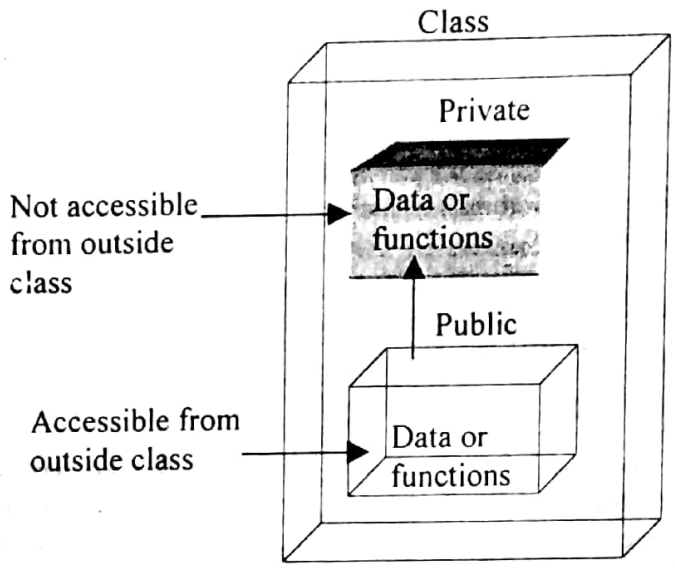


Figure 10: Private and Public

You can do anything with the private data within the function implementations that are a part of that class. You should keep in mind that the private data is not available outside the class to any program that might need to use the data member. Also, the private data of other classes is hidden and not available within the functions of this class.

It is legal to declare variables and functions in the private part, and additional variables and functions in the public part also. In most practical situations, variables are declared in only the private part and functions are declared in only the public part of a class definition.

1.5 Member functions

A message to an object corresponds to a function call. Messages can have parameters. In response to a message, the object executes a *method*, which is similar to a procedure body. It may or may not return a result. A method is a function contained within the class. In object oriented programming, we send messages instead of calling functions.

In C++, a function declared as a member of a class is called a *member function*. There are two member functions in the class `date` shown above:

- `setdate(int, int, int)`
- `print()`

Member functions are usually put in the public part of the class because they have to be called outside the class either in a program or in a function.

As a class contains not only a data member but also a function or methods, it must be defined before it is to be used. The declaration of a member function must define its return value as well

Durr-e-Danish Library
Scanned by CamScanner

as the list of its arguments. For example, the member function `void setdate(int, int, int)` has no return value or a void return value, and has three integer arguments.

Member functions can be more complex as they can have local variable, parameters etc. Member functions should not have the same name as a data member.

There are several benefits from restricting access to a data structure to an explicitly declared list of functions. Any error causing a date to take on an illegal value (for example, January 35, 1998) must be caused by code in a member function, so the first stage of debugging, localisation, is completed before the program is even run. Another advantage is that a potential user of such a type needs to only examine the definition of the member functions to learn to use it.

1.6 Using the class

Now that we know how a class is specified, let us see how objects are defined, and once defined, how their member functions are accessed. Let us consider the example listed below:

Example 1.

```
begin program

class exampleclass{                    // specify a class
  private:
     object_data is an integer;        // class data
  public:
     member_function1(parameter1, parameter2...)
          {assign value to object_data}
     member_function2()
          {display data}
};

main program
{
  exampleclass object1,object2;        //define the objects of class

     object1.member_function1(200);// call member function to
                                   //assign value 200 to
                                   // object_data

     object1.member_function2();   //call member function to
                                   // display data

     object2.member_function1(350);
     object2.member_function2();
```


Durr-e-Danish Library
Scanned by CamScanner

1.6.1 Defining Objects

The first statement in the main program above,

```
exampleclass object1,object2;
```

defines two objects, object1 and object2, of `class exampleclass`. Remember that the specification for the `class exampleclass` does not create any objects. It only describes how they will look when they are created, just as a structure specifier describes how a structure will look but does not create any structure variables. It is the definition that actually creates objects that can be used by the program. Defining an object is similar to defining a variable of any data type. When we define an object, space is set aside for it in memory.

1.6.2 Calling member functions

We communicate with objects by calling their member functions. The next two statements in the main program above call the member functions:

```
object1.member_function1(200);
object1.member_function2();
```

A member function is always called to act on a specific object, not on the class in general. A member function is associated with a specific object with the dot operator (the period). The general syntax for accessing a member function of a class is

```
class_object.function_member()
```

The syntax is similar to the way we refer to structure members, but the parentheses indicate that we are executing a member function rather than referring to a data item. The dot operator is called the *class member operator*. The first function call,

```
object1.member_function1(200);
```

assigns the value 200 to `object_data` of the `object1`. The second call causes the object to display its value. Similar functions are defined and executed for `object2`.

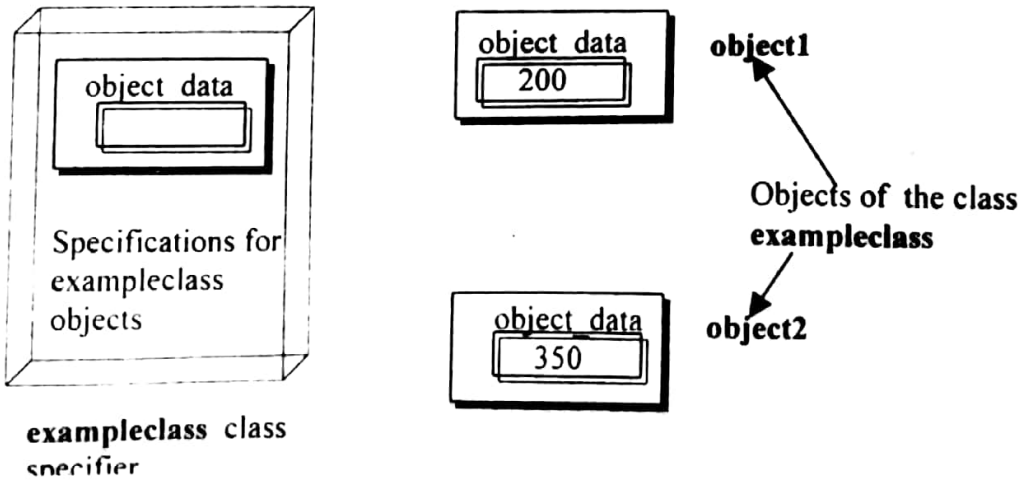


Figure 11: Two objects of a class with different values

Durr-e-Danish Library
Scanned by CamScanner

1.6.3 Passing and Returning Objects

Objects can be passed to a function and returned back just like normal variables. When an object is passed as an argument in a function, the compiler creates another object as a formal variable in the called function. It copies all the data members from the actual object. Taking a look at Example 1 above. Assume that we define a member function as given below:

```
int function1(exampleclass obj1)
{
      return object_data;
}
```

Formal argument

This function has an argument that is an object and it returns an integer value. The above function can be invoked using a function call such as,

```
int variable = object1.function1(object2);
```

actual argument

When a member function is called, it is given access to the object of which the function is a member. In the above function call `fucntion1` can access the data members of `object2` that is passed as an argument and also that of `object1`, which is the invoking object.

The data members of `object1` can be accessed in the function without any object name, for example, `object_data`. However, the name of the formal object `obj1` has to be used when the function has to access its data members, using the syntax `obj1.object_data`. Thus, passing an object to a member function of the same class results in two objects being accessible directly in the function. In the example the member function does access the data member of the class. The integer return value of the function is assigned to the integer variable.

Just as it is possible for a function to return an integer value it is also possible for the function to return an object. A return statement in a function is considered to initialise a variable of the returned type. The syntax remains the same as shown in the function call below:

```
exampleclass object3 = object1.function1(object2);
```

where the function would be defined as,

```
exampleclass fucntion1(exampleclass obj)
{
  exampleclass temp_object;
  .
  .
  return temp_object;
}
```

The function has a local object, `temp_object`, which is returned to the calling function.

Passing and returning of objects is not very efficient since it involves passing and returning a copy of the data members.

Durr-e-Danish Library

Scanned by CamScanner

1.7 Object-Oriented Languages

There are almost two dozen major object-oriented programming languages in use today. But the leading commercial object-oriented languages are far fewer in number. Some of the object-oriented languages that have been developed are:

- C++
- Smalltalk
- Eiffel
- CLOS
- Java

These languages vary in their support of object-oriented concepts. There is no single language that matches all styles and meets all needs.

1.7.1 An introduction to C++

C++ was developed by Bjarne Stroustrup at AT&T Bell Laboratories. C++ is derived from the C language. It is compatible with C (it is actually a superset), so that existing C code can be incorporated into C++ programs. The most important elements added to C to create C++ are concerned with classes, objects and object-oriented programming.

C++ programs are fast and efficient, qualities which helped make C an extremely popular programming language. However, it sacrifices some flexibility in order to remain efficient. C++ uses *compile-time binding*, which means that the programmer must specify the specific class of an object, or at the very least, the most general class that an object can belong to. This provides high run-time efficiency and small code size, but it trades off some of the power to reuse classes.

C++ has become so popular that most new C compilers are actually C/C++ compilers. However, to take full advantage of object-oriented programming, one must program and think in C++, not C. This can often be a major problem for experienced C programmers. Many programmers think they are coding in C++, but instead are only using a small part of the language's object-oriented power.

1.7.2 Smalltalk

Smalltalk is a pure object-oriented language. While C++ makes some practical compromises to ensure fast execution and small code size, Smalltalk makes none. It uses *run-time binding*, which means that nothing about the type of an object need be known before a Smalltalk program is run.

Smalltalk programs are considered by most to be significantly faster to develop than C++ programs. A rich class library that can be easily reused via inheritance is one reason for this. Another reason is Smalltalk's dynamic development environment. It is not explicitly compiled, like C++. This makes the development process more fluid, so that "what if" scenarios can be easily tried out, and classes definitions easily refined.

Being purely object-oriented, programmers will find it more difficult to adapt to programming in Smalltalk unlike C++. Smalltalk generally takes longer to master than C++. But most of this time is actually spent learning object-oriented methodology and techniques, rather than details of a particular programming language. In fact, Smalltalk is syntactically very simple, much more so than either C or C++.

Durr-e-Danish Library

Scanned by CamScanner

The Session in Brief

- Any application can be defined in terms of entities or objects so that the process replicates human thought process as closely as possible.
- Traditional programming techniques using data and functions do not model real-world concepts.
- Object-oriented programming allows for the analysis and design of an application in terms of entities or objects so that the process replicates the human thought process as closely as possible. In object-oriented programming, code and data are merged into a single indivisible thing -- an object.
- An object is a concept or thing with defined boundaries that is relevant to the problem we are dealing with. Each object has its own properties or characteristics that describe what it is or does. An object also executes some actions.
- Similar objects, objects with common properties, are grouped into a class. Each class describes a set of individual objects.
- Each object is said to be an instance of its class. Objects in a class have the same properties and common behaviour and common relationships to other objects.
- Abstraction consists of focussing on what an object is and does, before deciding how it should be implemented. Data abstraction is used to identify properties and methods of each object as relevant to the application at hand.
- In our daily lives, we use the concept of classes being divided into subclasses. The subclass inherits all the behaviour of the parent class.
- Providing access to an object only through its messages, while keeping the details private is called information hiding or encapsulation.
- All object-oriented languages try to make parts of programs easily reusable and extensible. Reusing existing code saves time and money, and increases a program's reliability.
- Polymorphism means that the same functions may behave differently on different classes.
- A structure contains one or more data items (called members) which are grouped together as a single unit. On the other hand, a class consists of not only data elements but also functions, which operate on the data elements.
- In a structure all elements are public by default.
- As one of the features of object-oriented programming is to prevent data from unrestricted access, the data members of a class are normally described in the private section. Private data or functions can only be accessed from within the class.
- Usually the functions in a class are public. The members and operations declared in the public section can be accessed by any function outside of the class.

Durr-e-Danish Library

Scanned by CamScanner

- In C++, a function declared as a member of a class is called a member function. Member functions are usually given the attributes of public because they have to be called outside the class either in a program or in a function.

- The declaration of a member function must define its return value as well as the list of its arguments.

- Defining an object is similar to defining a variable of any data type: space is set aside for it in memory. It is the definition that actually creates objects that can be used by the program.

- A member function is always called to act on a specific object, not on the class in general. A member function is associated with a specific object with the dot operator called the class member operator.

- Objects can be passed to a function and returned back just like normal variables. When an object is passed as an argument in a function, the compiler creates another object as a formal variable in the called function. A return statement in a function is considered to initialise a variable of the returned type.

- Some of the object-oriented languages that have been developed are:

  - C++
  - Smalltalk
  - Eiffel
  - CLOS
  - Java

- C++ is derived from the C language. The most important elements added to C to create C++ are concerned with classes, objects and object-oriented programming.

- Smalltalk is a pure object-oriented language. Smalltalk programs are considered by most to be significantly faster to develop than C++ programs. It is not explicitly compiled, like C++.

- Smalltalk generally takes longer to master than C++. But Smalltalk is syntactically very simple, much more so than either C or C++.

Durr-e-Danish Library

Scanned by CamScanner

Check Your Progress

1. In the object oriented way of thinking, software can be organised as a collection of objects that combine both behaviour and Data Structure.

2. Combining data and functions together as a single entity is called Encapsulation

3. Camel is to animal as object is to
   a. a member function
   b. a class
   c. an operator
   d. a data item

4. An object is an instance of the class. **True/~~False~~**

5. Protecting data from access by unauthorised functions is called Encapsulation

6. In structures all functions and data, by default, are Public but private in classes.

7. In a class specifier, data or functions designated private are accessible
   a. to any function in the program
   b. to member functions of that class
   c. only to public members of the class
   d. only if you know the password

8. In C++, a function contained within a class is called Member function

9. Data items in a class must be private. **~~True~~/False**

10. To use a member function, the Dot connects the object name and the member function.

11. Sending a message to an object is the same as Calling as one of member function

12. C++ is considered a Superset of C.

Durr-e-Danish Library

Scanned by CamScanner

Do it Yourself

1. Prepare a list of objects that would be part of an Automobile system.

2. Discuss what the objects in the following list have in common:
   Bicycle, sailboat, car, truck, aeroplane, glider, motorcycle, horse

3. Name the classes the following objects can be grouped into:
   a. File, directory, filename, ASCII file, executable file
   b. Expression, constant, variable, function, statement, arithmetic operator
   c. Sink, freezer, refrigerator, bread, cheese, door, cabinet, mop

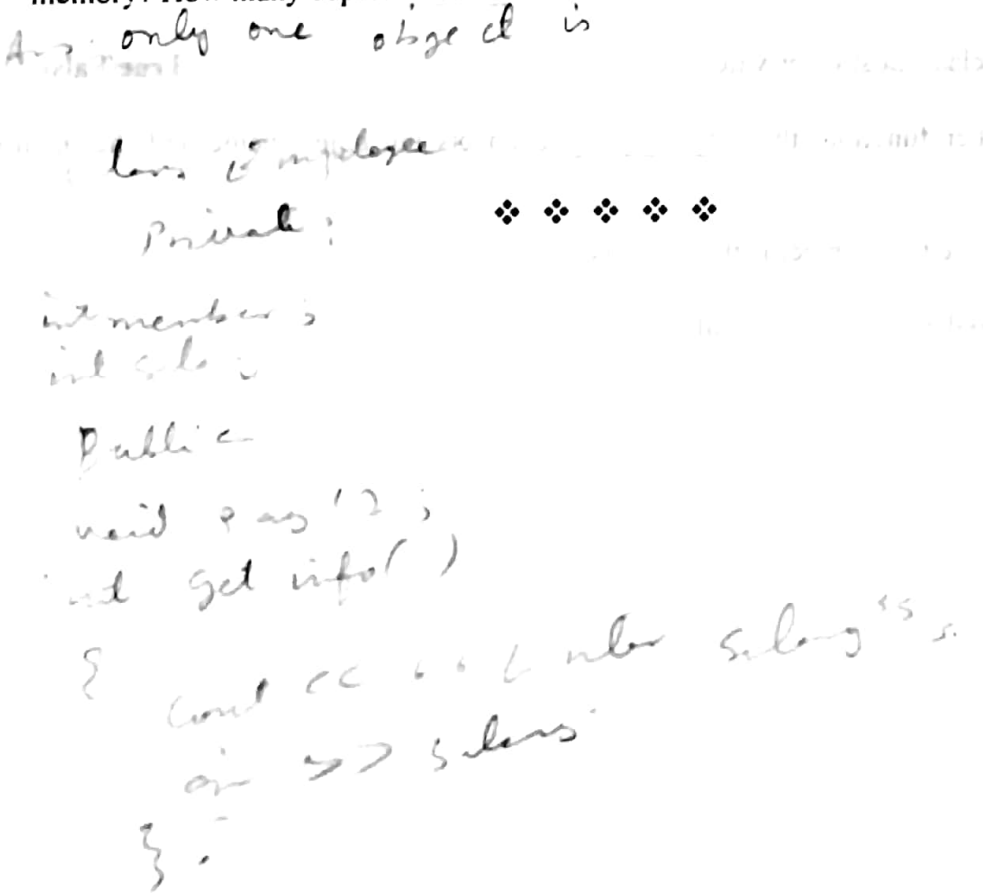
4. Write a class specifier, in C++, that creates a class called Employee with one private data member, salary, of type int and one public function whose declaration is void pay().

5. Write a statement that defines an object called Officer of the Employee class described in question 4.

6. Write a statement that executes the pay() function using the Officer object, as described in question 4 and 5.

7. Write a member function called getinfo() for the Employee class described in question 4 and 5. This function should return the value of the salary data.

8. If three objects of a class are defined, how many copies of that class's data items are stored in memory? How many copies of its member functions?



Durr-e-Danish Library
Scanned by CamScanner

Session Objectives:

At the end of this session, the student will be able to -

- *Use the scope resolution operator*
- *Use dynamic memory allocation with*
  - *New*
  - *Delete*
- *Use pointers to objects*
- *Define and use Constructors*
- *Define and use Destructors*
- *Define the "Const" keyword*
- *Define and use the "this" pointer*
- *Describe how objects and functions are arranged in memory*
  - *Static Data Members*
  - *Static member Functions*
- *Describe type conversions using*
  - *Converting by assignment*
  - *Type casting*

2.1 Scope resolution operator

In the earlier session we have seen the definition of a class, the objects and the member functions which are part of the class. So far we have seen that member functions and member variables were defined in the class specifier. When more member functions are added to a class specifier it becomes bulky and difficult to handle. At such times it is useful to keep only the function declarations within the class specifier and define the functions outside it. To tell the compiler that a function is a member of a class it has to be declared within the class specifier. It can be defined outside the class specifier using a *scope resolution operator* :: (double colon symbol) with the function definition.

The general syntax for using member functions outside a class specifier is:

```
return_type class_name::member_functions(arg1, arg2,. . .,argn);
```

- The type of member function arguments must exactly match with the type declared in the class specifier.

The use of the scope operator is important for defining the member functions outside the class declaration. The left-hand operator of :: must be the name of the class. Only the scope operator identifies the function as a member of a particular class. Without the scope operator, the function definition would create an ordinary function.

Let us look at a member function `sum()` declared in a class `sample`. The following example shows how the member function is defined outside the class specifier.

Durr-e-Danish Library

Scanned by CamScanner

Example1

```
class sample{
      .
      .
      .
      public:
      int sum();        // declaration of the function
};

int sample::sum()       // definition of member function
{
      .                 // code
      .
      return value of type int;
}
```

The scope resolution operator is also used to refer to global variable names in cases where a global variable and a local variable share the same name. The syntax used is:

```
::global_variable
```

By letting you distinguish between variables with the same name, the scope resolution operator gives you more freedom in naming variables. However, this feature should not be overused. If two variables have different purposes, their names should reflect the difference.

2.2 Dynamic memory allocation

Unless a programmer specifies otherwise, an object is created when its definition appears in a program and it is destroyed when its name goes out of scope or the program terminates. Often it is useful to create a new object that will exist only as long as it is needed.

In C++ the operator `new` creates such objects and the operator `delete` can be used to destroy them later. Objects allocated by `new` and `delete` are said to be on the free store.

2.2.1 New

The `new` operator is used to create a memory space for an object of a class. The `new` operator returns a pointer to the object created. The general syntax of the new operator is:

```
data_type pointer_variable  = new data_type;
```

where `data_type` can be a `short int, float, char, array` or even class objects.

For example,

```
int *p;           //pointer to integer type
float *f;         //pointer to a float type
p = new int;      //allocates memory for an integer
f = new float;    //allocates memory for a float
```


Durr-e-Danish Library
Scanned by CamScanner

If the call to new is successful, it returns a pointer to the space that is allocated. Otherwise it returns zero if the space is not available or if some error is detected.

The same syntax that is used for allocating memory for a basic data type can be used to allocate memory for an object. For example, let us suppose there is a class Student.

```
Student *stu_ptr;          //pointer to an object of type Student
stu_ptr = new Student;     //points to new Student object
```

The above statements create an object of type Student using the new operator and return a pointer to it, which is assigned to stu_ptr.

- The new operator is similar to the malloc() function used in C. The new operator is superior in that it returns a pointer to the appropriate data type, while malloc()'s pointer must be cast to the appropriate type.

2.2.2 Delete

If we allocate memory when we create an object, we need to de-allocate the memory when the object is no longer needed. An object created by new exists until it is explicitly destroyed by delete. The space it occupied can then be reused. The general syntax of the delete operator is:

```
delete pointer_variable;
```

The following code shows the simplest use of new and delete.

Example 2

```
int *ptr;
ptr = new int;
*ptr = 12;
cout << *ptr;
delete ptr;
```

In the above example a pointer to an integer, ptr, is made to point to the memory space created by new. The value of 12 is assigned to the integer, which is pointed to by ptr. The memory allocated is de-allocated using delete. As the new operator returns a pointer to the object being created, the delete operator must define a pointer name only but not with data type.

Suppose new is used in a function and the function uses a pointer local variable to point to this memory. When the function terminates, the local variable, i.e. the pointer will be destroyed but the memory will be left as it is, taking up space that is inaccessible to the rest of the program. Thus it is always a good practice to delete memory when you are through with it. Be careful that you do not use pointers to memory that has been deleted.

It is also possible to allocate blocks consisting of arrays of varying length using a similar technique. Note the use of delete [] for deleting the array:

Durr-e-Danish Library

Scanned by CamScanner

Example 3

```
int *ptr;
ptr = new int[100];
delete [] ptr;
```

In the above example we see that memory space is allocated for 100 integer objects. Any time you allocate an array of objects using `new`, you must use `[]` in the `delete` statement. This syntax is necessary because there is no syntactic difference between a pointer to an object and a pointer to an array of objects. So if you were to use delete without the brackets you would not be deleting the full array.

The standard functions which are used in C for dynamic memory management, `malloc()`,and `free()`, are also available for use in C++ and can be used in the same manner they were used in C. If you are designing and coding a new program you should use the `new` and `delete` constructs because they are a built-in part of the language rather than an add-on feature and are therefore more efficient.

Note that it is an error to `delete` a variable that has been `malloc`'ed and it is an error to `free` a variable that was allocated with `new`.

2.3 Pointers to objects

Pointers can point to objects as well as to simple data types. Declaring a pointer to an object of a particular class is the same as declaring a pointer to a variable of any data type. We have seen examples of objects defined and given a name such as:

```
date today;
```

where an object called `today` is an instance of the class `date`.

At the time we write the program we do not know how many objects we want to create. We have seen that in such a case we can use `new` to create objects while the program is running. As we have seen, `new` returns a pointer to an object. For example,

```
date *today_ptr;       //pointer to an object of type date
today_ptr = new date; //points to the new date object
```

This creates an object of type `date` using the `new` operator and returns a pointer to it that is assigned to `today_ptr`.

We need to have a means to refer to any member functions in the object pointed to by `today_ptr`. The dot (.) membership-access operator will not work in this case. The dot operator requires the identifier on its left to be an object. Since `today_ptr` is a pointer to an object, we need another syntax. This is done using the arrow operator (`->`). For example, if a member function called `getdate()` was declared in the class `date`, we can refer to it in the following manner:

```
today_ptr->getdate();
```


Durr-e-Danish Library
Scanned by CamScanner

2.4 Constructors

In the earlier session we have seen how data items in an object are assigned values using member functions. Since it is nowhere stated that an object must be initialised, a programmer can forget to do so or may do so twice. A better approach is to allow the programmer to declare a function with the explicit purpose of initialising objects.

Definition:

A *constructor* is a special member function for automatic initialisation of an object.

Whenever an object is created, the special member function, the constructor, will be executed.

The constructor is distinguished from all other member functions by having the same name as the class it belongs to. You can declare and define constructors within the class, or declare them within the class and define them outside just as any other member functions. The general syntax of the constructor function in C++ is:

```
class username {
      .
      .
      public:
      username();          //constructor
};

username::username()       //constructor defined outside the class
{     .                    //code
      .
}
```

No return type is used for constructors. The constructor is called automatically by the system. Therefore, a return value would make no sense. Constructors are also invoked when local or temporary objects of a class are created.

When a class has a constructor, all objects of that class will be initialised. It may sometimes be necessary to provide several ways of initialising a class object. This can be done by providing several constructors. A *default constructor* is a constructor that does not have any arguments. Even if it is not explicitly written in a program the compiler automatically generates it. If a class has a constructor with arguments, that constructor will be used when an object of that type is initialised. For example,

Example 4

```
class date{
      int month, day, year;
public:
      date()                                  //default constructor
      {day=1; month=1; year=1999;}
      date(int x)                             //only day is specified
      {day=x; month=1; year=1999;}
      date(int x, int y, int z)               //day month year
```


Durr-e-Danish Library

Scanned by CamScanner

```
        {day=x; month=y; year=z;}
};
```

In Example 4, the three constructors for the class `date` differ in the number of arguments that each takes. In this example the default values are given as 1/1/1999 to explain how the values are assigned. The default constructor can be used to initialise the date object with the current date or any other date as required by the program.

As long as the constructors differ sufficiently in their argument types the compiler can select the correct one for each use, as shown in the examples given below. Each example calls a different constructor depending on the value of the argument.

```
date now;                    //default initialised
date today(4);               //constructor with one argument used
date all(23,3,1998);         //constructor with three arguments used
```

Once we define one constructor, we must also define the default constructor. When designing a class the programmer may often be tempted to add features just because somebody might want them at a later date. Deciding on what features are really needed, and including only those, is an important part of programming. It may require more thought but usually leads to smaller programs that are easier to understand.

2.5 Destructors

A constructor is used to create an object and initialise data members with the parameters passed to it. This object remains in existence following the scope rules just like any variable. Another special member function called the *destructor* is called when an object goes out of existence, i.e. when its scope is over. When an object is about to be automatically destroyed, its destructor, if one exists, is called automatically.

Definition:

A *destructor* is a member function that is called automatically when an object is destroyed.

A constructor can be explicitly called at the time of initialisation, as we have seen earlier, but a destructor cannot be directly called from the class. The compiler itself generates a call to a destructor when an object expires.

A destructor also has the same name as the class but with a tilde (~) before the class name. The general syntax for the destructor in C++ is:

```
class username {


  public:
      ~username();      //destructor
};
```

Like constructors, destructors have no return type. They also take no arguments. The assumption is that there is only one way to destroy an object.

Durr-e-Danish Library
Scanned by CamScanner

The most common use of destructors is to de-allocate memory that was allocated for the object by the constructor using the new operator. Let us look at an example.

Example 5

```
class String{
      private:
            char *str;
      public:
            String(char *s){              //constructor
                  int length = strlen(s);
                  str = new char[length+1];
                  strcpy(str,s);
            }
            ~String() {delete[] str;}   //destructor
   };
```

In the above example a string is allocated memory in the constructor, using the `new` operator. The length of a string is calculated and accordingly space is allocated. The memory allocated has to be de-allocated using a `delete` operator. Since the string created with new is an array of characters we need to use `[]` with `delete` in the destructor.

It is interesting to note that if a constructor is used for an object that is declared before the main() program, as a global variable, the constructor will actually be executed before the execution of the main () program. Similarly, if a destructor is defined for such a variable, it will execute following the completion of execution of the main () program. This will not adversely affect your programs, but it is interesting to make note of.

2.6 The Const keyword

A constant is an entity whose value does not change during the execution of a program. This is useful at times when we know that a certain value will remain the same in a program. For example you may want to specify the value of PI in a program. There can be only one value for PI used throughout the program. Any attempt to change the value of this constant will result in an error.

The keyword `const` can be added to the declaration of an object to make that object a constant rather than a variable. In C++, the word `const` is applied to normal variable declarations as shown below:

```
const int num=100;
```

The `int num=100;` portion is formatted exactly the same way as a normal declaration. The word `const` in front of it simply defines that the variable num cannot be subsequently changed. A constant cannot be assigned to, so it must be initialised. Declaring something `const` ensures that its value cannot change within its scope:

```
num=200;        //error
num++;          //error
```


Durr-e-Danish Library
Scanned by CamScanner

1) only pointer can point to any const unless (if it holds type is integer) but doesn't allows change the value of const it.
2) It allows to change the value but doesn't allow to point anywhere

➢ Const qualifier with pointers

The const qualifier can be put to some other uses involving pointers. When we use const with a pointer, there are two objects involved. One is the pointer itself and the other is object pointed to. Prefixing a declaration of a pointer with const makes the object, but not the pointer, a constant. Read the following examples carefully:

Example 6

pointer of const integer type
object pointed to

constant integer

```
int num =10;
const int *iptr = &num;
*iptr = 25;                    //error, iptr points to a constant
num = 25;                      //valid, num is not constant

int xyz = 200;
iptr  = &xyz;                  //iptr can point anywhere else
*iptr = 305;                   //error
```

In this example, iptr points to a constant integer. It cannot be used to change the contents of the const integer. However, it can be made to point somewhere else but only to another const integer.

It is also possible to declare a pointer itself as a constant rather than the object pointed to. To do this the operator *const is used. For example;

Example 7

constant pointer

```
int a1 = 777;
int *const ptr = &a1;          //constant pointer to integer
*ptr = 66;                     //valid; does not point to const
int a2 = 45;
ptr = &a2;                     //error, ptr is a constant
```

In the above example, ptr is a constant pointer to an integer variable. It cannot be made to point to another integer since it is constant. However it can be used to change the contents of the integer it points to.

for arguments

The const keyword can also be used in parameter lists to specify the valid usage of a parameter. This feature is particularly useful in function arguments. When you declare a pointer argument as constant, the function cannot modify the object pointed to. For example:

Example 8

Same as e.g(8)
Const character

```
void func(const char *str)
{
  str  = "hello";      //valid
  *str = 'A';          //error; cannot modify what is pointed to
}
```


Durr-e-Danish Library
Scanned by CamScanner

In the above example the function cannot modify the contents of the string that `str` points to. However, `str` can be made to point to another string `"hello"`.

Example 9

```
void func1(const int index)
{
  index = 5;                    //error; cannot modify a constant
}
```

In the function `func1` shown in the example above, the integer `index` that is passed as an argument to the function is a constant. Its value cannot be modified in the function.

2.7 This pointer

A class can have many objects. The name of the object is not required for accessing members of an object in a member function. How does the compiler understand which of the object's members is being referred to in a member function? Obviously, the one that was used to invoke that member function. To make explicit reference to an object's members, the member functions of every object have access to a special pointer named `this`.

The keyword `this` gives the address of the object, which was used to invoke the member function. Whenever a member function is called, the compiler assigns the address of the object which invoked the function, to the `this` pointer.

Thus, the `this` pointer can be used like any other pointer to an object. It can be used to access the members of the object it points to with the use of the arrow operator. For example:

```
this->age = 5;          //accessing the data member age
this->getdata();        //invoking a member function of the class
```

The example below shows the use of `this`.

Example 10

```
class Person{
      private:
            int age;
      public:
            void display();
};

void Person :: display();
{
      this->age = 25;         // same as age=25
      cout<<this->age;        // same as cout<<age
};

void main()
```


Durr-e-Danish Library

Scanned by CamScanner

```
{       Person Jack;
        Jack.display();
}
```

Notice that the member function `display()` accesses the variable age as this->age. This is exactly the same as referring to age directly.

A more practical use for `this` is in returning values from member functions. When an object is used as a local variable in a function it is destroyed when the function completes execution. Thus, it is not possible to return by reference an object that is local to a function. However, the object that invokes the member function is only destroyed at the termination of a program. Using a `this` pointer it is possible to return by reference the object that invokes a member function instead of a local object. Later we will see how to use `this` when we want to return by reference the object from the member function.

2.8 Objects and functions in memory

Each object has its own copy of the data members of the class. However, all the objects in a given class use the same member functions. The member functions are created and placed in memory only once - when they are defined in the class specifier. The data items however will hold different values, so there must be a separate instance of each data item in each object. Data is therefore placed in memory when each object is defined, so there is a set for each object. Figure 1 shows how this looks.

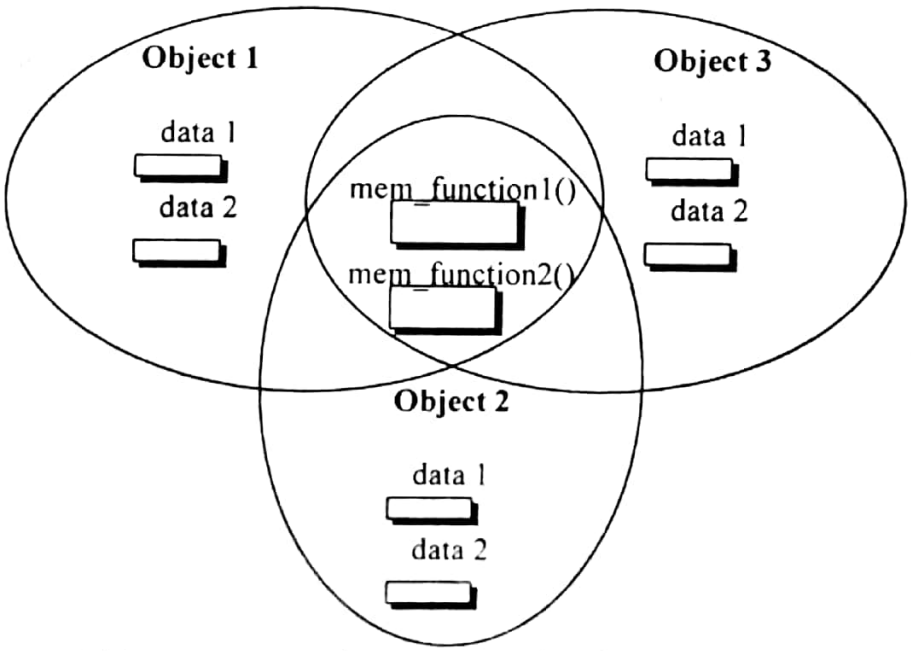


Figure 1: Objects, data members and member functions in memory

2.8.1 Static Data Members

A static data item is useful when all objects of the same class must share a common item of information. If a data item in a class is defined as *static*, then only one such item is created for the entire class, no matter how many objects there are. A member variable defined as static has

Durr-e-Danish Library
Scanned by CamScanner

similar characteristics to a normal static variable. It is only visible within the class, but its lifetime is through the entire program.

The general syntax for a static data member is:

```
static data_type variable;
```

As an example, suppose an object needed to know how many other objects of its class were in the program. For example, in a road-race game a racing car might want to know how many cars were still in the race. If a static variable count were to be included as a member of the class, all the objects would have access to this variable. All the racing cars would see the same variable count. This is how it would be declared in a class specifier:

Example 11

```
class race_cars{
  private:
      static int count:
      int car_number;
      char name[30];
  public:
      race_cars(){count++;}          //constructor to increment count
      ~race_cars(){count--;}         //destructor to decrement count

};
int race_cars::count;                //initialise count before main()
```

The static data member should be created and initialised before the `main()` program begins.

Every time that an object of type `race_cars` is created the constructor adds one to the static variable `count`. Each object will see the same value for `count`. Here is a graphical representation of the same problem:

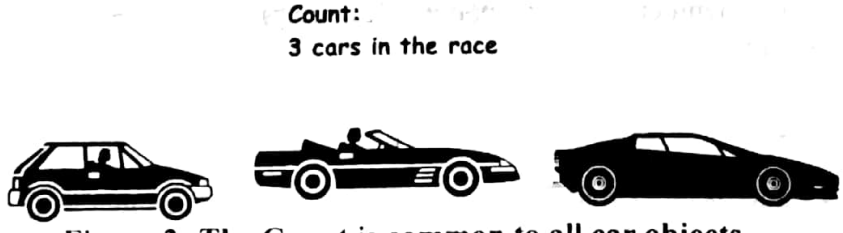


Figure 2: The Count is common to all car objects

The access rule of a static member is the same as for other data members. If a static member is declared as a private category of the class, the non-member functions cannot access it. If it is declared as public, then any member of the class can access.

Durr-e-Danish Library

Scanned by CamScanner

Whenever a static data member is declared it has only a single copy that will be shared by all instances of a class. This means that the static member becomes a global data for the class. The use of static data members can considerably reduce the need for global variables.

2.8.2 Static Member Functions

Declaring a member static restricts its scope and makes it independent of the individual objects of the class. This feature is as useful for member functions as for data members. A static member function can manipulate only on the static data member of the class. The static member function acts as global for members of its class without affecting the rest of the program.

A static member function is not part of the objects of a class. It does not have a `this` pointer. To invoke a static member function the syntax `object.function()` is not used. It can be called directly by using the class name and the scope resolution operator. Let us look at an example that uses a static data member and a static member function.

Example 12

```
class alpha{
    private:
      static int count;              //static data member
    public:
      alpha(){count++;}              //constructor increments count
      static void display_count() //static member function
      { cout<<count; }
};

void main()
{
      alpha::display_count();     //before any object is created
      alpha  obj1, obj2, obj3;
      alpha::display_count();     //after three objects created
}
```

In the above example the constructor increments a static data member `count` each time an object is created. As a result of the first call to the static member function in `main()` the value of `count` that is displayed in the output will be 0. Even when no object has been created we can directly call the static member using the name of the class and the scope-resolution operator as shown in the statement:

```
alpha::display_count();
```

This is because a static member function is considered to be independent of the objects of a class. You can use a static member function to query an object to find out which object it is. This is sometimes useful in debugging a program, among other situations.

Durr-e-Danish Library

Scanned by CamScanner

2.9 Type Conversions

Occasionally you may find it necessary to use expressions, which involve different data types. Type conversion is done to convert a variable of a declared type to some other required type. Consider the example given below:

Example 13

```
int count = 7;
float average = 15.5;
double totalsum = count * average;
```

A variable, `count`, of type `int` is multiplied by a variable, `average`, of type `float` to give the result, `totalsum`, which is of type `double`. Such an expression would be compiled without error. The compiler considers it normal that you would want to perform arithmetic operations on numbers of different types.

Converting data types can be done in two ways:

- Converting by assignment
- Type casting

2.9.1 Converting by assignment

Converting by assignment is a typical way of converting a value from one data to another using the assignment operator (=). Data types are considered " higher" or "lower" in the order shown:

Highest lowest

```
Long double >double>float>long>int>char
```

When two operands of different types appear in the same expression the lower-type variable is converted to the type of the higher-type variable. This is an implicit or automatic conversion performed by the compiler. These conversions take place invisibly and usually you do not need to think too much about them. This means that we can convert a value from one type to another just by assigning a `float` variable's value to a `double` variable, or a `char` variable's value to an `int` variable. For example,

Example 14

```
int   xin, yin;
float aft, bft;

xin = 5;                  //integer assigned a value
aft = 31.0135;            //float assigned a value

yin = aft;                //float assigned to integer
bft = 21.5/xin;           //integer converted to float
```

In the above two statements, the first one assigns a float variable `aft` to an integer variable `yin`. The value in `yin` will be 31. The value in `yin` gets truncated since the conversion is taking place

Durr-e-Danish Library
Scanned by CamScanner

from a `float`, which is a higher-order type, to an `int`, which is a lower-order type. In the second statement, the integer `xin` is first converted to a `float` and then used in the expression. Here the integer variable `xin` of lower-order type is being converted to a higher-order type. The resultant value assigned to the floating-point variable `bft` will be 4.3.

Converting by assignment is not recommended since it will truncate the fractional or real parts of a value. One may not get the desired results if this happens. To overcome this problem, it is better to explicitly convert one data type to another using the type casting.

2.9.2 Type Casting

In C++ the term casting applies to data conversions that are specified by the programmer. Sometimes we need to convert a value from one type to another where the compiler will not do it automatically. A typical use of a cast is in forcing a division to return a `float` when both operands are of the type `int`. For example:

```
result = float(21)/5;
```

In the example, we want `result` to show a `float` value. Without the cast operation, when 21 is divided by 5, a truncated integer division is performed. The cast operation explicitly converts 21 to a `float` and causes the division to be carried out as a floating-point division. Let us look at some more examples:

```
char chs;
int x, y, z;
float aft;

x = int(chs);
```

The above statement forces a character variable to an integer.

```
aft = float(y*z);
```

The expression `(y*z)` is an integer data type and it is forced to convert to a floating number.

You can use another syntax for casts:

```
(int)chs;
```

with the parentheses around the type rather than around the variable. The first approach, called *functional notation*, is preferred in C++ because it is similar to the way other parts of C++, such as functions are written.

Durr-e-Danish Library

Scanned by CamScanner

- The scope operator :: is important for defining the member functions outside the class declaration. Only the scope operator identifies the function as a member of a particular class. The left-hand operator of :: must be the name of the class.

- The scope resolution operator is also used to refer to global variable names in cases where a global variable and a local variable share the same name.

- A constructor is a special member function, with the same name as its class, which is automatically initialised whenever an object is created. A constructor has no return type but can take arguments. You can have constructors that take arguments of different types so an object can be initialised in different ways.

- A destructor is a member function with the same name as its class but preceded by a tilde (~). It is called when an object is destroyed. A destructor takes no arguments and has no return value.

- The `new` operator is used to create a memory space for an object of a class and returns a pointer to the object created. The `delete` operator releases memory obtained with new.

- Pointers can point to objects as well as to simple data types. Declaring a pointer to an object of a particular class is the same as declaring a pointer to a variable of any data type.

- When a pointer points to an object, members of the object's classes can be accessed using the access operator ->.

- The keyword `const` can be added to the declaration of an object to make that object a constant rather than a variable.

- To make explicit reference to an object's members, the member functions of every object have access to a special pointer named `this`. The word `this` is defined within any object as being a pointer to the object in which it is contained.

- The `this` pointer can be treated like any other pointer to an object. It can be used to access the data in the object it points to with the use of the arrow operator.

- There is a separate copy of the data members for each object that is created from a class, but only one copy of a class's members. A data item can be restricted to a single instance for all objects of a class by making it static.

- A static member function acts as global for members of its class without affecting the rest of the program.

- In expressions involving variables of different data types, variables can be converted form one type to another or cast to another type by the programmer.

Durr-e-Danish Library

Scanned by CamScanner

Check Your Progress

1. The Scope resolution operator is important for defining member functions outside the class declaration.

2. In a class you can have more than one constructor with the same name. True/~~False~~

3. A default Constructor initialises data members with no arguments.

4. Identify which of the following is a destructor for a class `time`.
   a. `time(int, int);`
   b. `~time();`
   c. `~time(int, time);`
   d. `time::time();`

5. The `new` operator
   a. returns a pointer to an object created
   b. creates an object called new
   c. obtains memory for a new class
   d. tells how much memory is allocated for an object

6. The `delete` operator must define a pointer name with its data type. ~~True~~/False

7. To refer to any member function in an object pointed to by a pointer the this arrow operator is used.

8. Any attempt to modify the value of a variable defined as Const will result in an error.

9. If, within a class, `day` is a member variable the statement `this.day = 28;` assigns 28 to `day`. ~~True~~/False

10. A static is useful when all objects of the class must share the same data.

11. It is correct to use variables of different data types in the same arithmetic expression. True/~~False~~

Durr-e-Danish Library

Scanned by CamScanner

Do it Yourself

1. Assume a member function, `get_num()`, with return value of type `int` is declared inside a class called `Calculator`. Write a typical class specifier and declare the member function. Also write the definition of the function outside the class specifier.

2. Add a private data member called `number` of type `int` to the class `Calculator` in question 1. Write a constructor that initialises to 0 the number data. Assume the constructor is defined within the class specifier.

3. Given a pointer `ptr` that points to an object of type `first`, write an expression that executes the `get_num()` member function.

4. Given the program:

```
#include<iostream.h>
void main()
{
  cout<< "A program with a difference\n";
}
```

modify it to produce the following output using a class called `Check`.

```
Starting up
A program with a difference
Close program
```

5. Write a program to use a String class with a member function that converts a string from lower case to upper case. You can make use of standard library functions like toupper() to convert single characters to upper case. You can find this function in the header file CTYPE.H.

6. Write a program for the String class in which you can define a constant string, which holds the value "Hello". The program should ask the user to input a name. The name should be concatenated with the constant string and displayed as "Hello My_Name". Use new and delete operators to allocate and deallocate memory for strings.

Hint: Assign the constant to another data member of char* type before concatenating strings.

❖ ❖ ❖ ❖ ❖

Durr-e-Danish Library

Scanned by CamScanner

This page has been intentionally left blank

Durr-e-Danish Library

Scanned by CamScanner

Session Objectives:

At the end of this session, the student will be able to -

- *Understand the concept of functions with default arguments*
- *Define and use Friend functions*
  - *advantages*
  - *disadvantage*
  - *friend classes*
- *Describe function overloading*
  - *various data types*
  - *different number of arguments*
  - *describe the scope for function overloading*
- *Explain the use of reference arguments*
  - *passing references to functions*
  - *returning references from functions*
- *Define and use Inline functions*

3.1 Functions with default arguments

In any program when we need to perform any actions using expressions or statements we call a function to do it. When we define a function we are actually specifying how an operation is to be done.

A function declaration gives the name of the function, the type of the value returned (if any) by the function, and the number and types of the arguments that must be supplied in a call of the function. A function declaration may or may not contain argument names. In C++, it is possible to call a function without specifying all its arguments. Of course, this will not work on just any function. The function declaration must provide default values for those arguments that are not specified.

In the function declaration, the default values are given. Whenever a call is made to a function without specifying an argument, the program will automatically assign values to the parameters from the default declaration. Let us look at an example. Consider the function declaration

```
void func(int = 1, int = 3, char = '*');  //prototype declaration
```

The default argument follows an equal sign that is placed directly after the type name. You can also use variable names, as shown below:

```
void func(int num1,int num2 = 3,char ch = '*');
```

In the function declaration, once an argument is given a default value in the list of formal arguments, all of the remaining must have default values also. It is not possible to leave out any of the values in the middle of the list. Only the trailing values can be defaulted. As for example,

Durr-e-Danish Library

Scanned by CamScanner

```
void func(int num1=2,int num2, char ch='+');      //error
```

When the function is called, if any argument is missing it is assumed to be the last argument. The following calls to the function `func` declared above, are all considered valid except for the last one.

```
func(2,13,'+');
func(1);          //default values for second and third arguments
func(2,25);       //default value for third argument
func();           // default values for all three arguments
func(2,,'+');     //invalid
```

The missing arguments for the function call must be those at the end of the argument list. If you leave out any arguments in the middle the compiler would not know what you are referring to and will indicate an error. The default values must be of the correct types or the compiler will issue an error.

The default values can be given in either the prototype or the function definition header, but not in both. As a matter of style, it is highly recommended that the default values be given in the prototype declaration rather than in the function definition.

➢ **Advantages**

- Default arguments are useful if you want to use arguments, which will almost always have the same value in a function.
- They are also useful when, after a program is written, the programmer decides to increase the capability of a function by adding an argument. In such a case, the existing function calls can continue to use the old number of arguments, while new function calls can use more.

3.2 Friend Functions

The basic object-oriented concepts of data hiding and encapsulation are implemented by restricting non-member functions from accessing an object's private data. The basic policy is, if you are not a member then you cannot get in. Private data values cannot be read or written to by non-member functions. Sometimes this rigid separation can be inconvenient.

Durr-e-Danish Library
Scanned by CamScanner

Imagine that you want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private data. Of course this is not possible since a function cannot be a member of two classes. What we need is a means to allow a function access to the private part of a class without requiring membership. A non-member function that is allowed access to the private part of a class is called a *friend* of the class.

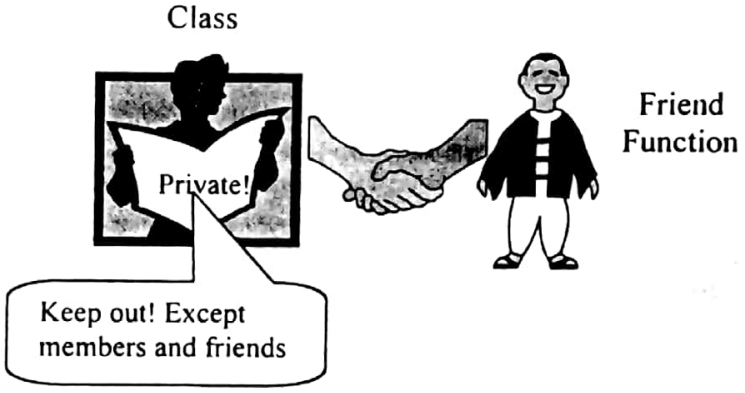


Figure 1: A friend function can access the private part of a class

A function is made a friend of a class by a friend declaration in that class. For example:

Example 1

```
class person{
      .
      .
      .
      public:
            void getdata();
            friend void display(person abc);
};
void display(person abc) //friend function without :: operator
{//...  some code...}
```

The keyword friend is not repeated in the function definition.

The friend function in the above example could perhaps have done the same operation if it were a member function of the class. However, if the same function needed to access objects from different classes it would be most useful to make it a friend of the different classes. Let us look at an example.

Example 2

```
class Teacher;                              //forward declaration
class Student{
     private:
           int st_data;
     .
     .
     .
```


Durr-e-Danish Library
Scanned by CamScanner

```
    public:
        void getstuddata();
        friend void display(Student abc, Teacher xyz);
};
class Teacher{
    private:
        int th_data;
    .
    .
    public:
        void getteachdata();
        friend void display(Student abc, Teacher xyz);
};
void display(Student abc, Teacher xyz)
{//...  some code...}
```

In the above example a friend function `display()` accesses members of two classes. The friend function is declared in the class specifier of both the classes although it does not belong to either. To access the private members of the classes the name of the data member has to be prefixed with the name of the object along with the dot operator. For example, in the definition of the friend function given above, to access the data member of the object `abc` the syntax would be:

```
abc.st_data
```

In Example 2, the class `Teacher` has been declared before the class `Student`. This is because a class cannot be referred to until it has been declared. The class `Teacher` is referred to in the declaration of the friend function in class `Student`. Therefore, it has been declared at the beginning of the program. This is called a forward declaration.

There are some features of a friend function that are worth remembering.

- There is nothing special about a friend function apart from its right to access the private part of a class.
- A friend function does not have a `this` pointer.
- The friend declaration is unaffected by its location in the class. It can be placed in either the private or public part of a class specifier.
- The definition of a friend function does not require the class name with the scope resolution operator prefixed to it, as it is required for a class member function.

Only when a function accesses private members of two or more classes directly, it has to be declared as a friend function. Otherwise public members of a class can be accessed directly by any function. There is some controversy regarding the use of friend functions. On one hand, friend functions increase flexibility in programming, on the other they are against the principles of object-oriented programming. However, this breach of integrity in the coding can be controlled to some extent.

A friend function has to be declared in the class whose data it will access. This cannot be done if the source code is not available to a programmer. If the source code is available, then existing classes should not be modified as far as possible. If you intend to use friend functions the class should be designed as such, right from the beginning. Even then, friend functions are not a tidy

Durr-e-Danish Library
Scanned by CamScanner

concept to work with because it involves a lot of intermingling with different classes. Sometimes they are unavoidable, but excessive use of friend functions over many classes also suggests a poorly designed program structure.

➢ **Advantages:**

Friend functions provide a degree of freedom in the interface design options. Member functions and `friend` functions are equally privileged. The major difference is that a `friend` function is called like `func(xobject)`, while a member function is called like `xobject.func()`. The ability to choose between member functions (`xobject.func()`) and `friend` functions (`func(xobject)`) allows a designer to select the syntax that is considered most readable, which lowers maintenance costs.

3.2.1 Friend classes

We can declare a single member function or a few member functions or a whole class as a friend of another class. Now, you may wonder when we would want to declare a whole class as a friend. When all or most of the functions of a particular class have to gain access to your class, you can consider allowing the whole class the friend privileges. Instead of declaring each of the member functions of the class as friends, you can save time by declaring the whole class as a friend.

Let us look at an example where the member function of one class is a friend of another.

Example 3

```
class beta;                          //forward declaration
class alpha{
  private:
      int a_data;
  public:
      alpha(){a.data = 10;}
      void display(beta);
};
class beta{
  private:
      int b_data;
  public:
      beta(){b_data = 20;}
      friend void alpha::display(beta bb);
};
void alpha::display(beta bb);
{
      cout<<"\n data of beta ="<<bb.b_data;
      cout<<"\n data of alpha ="<<a_data;
}
void main()
{
      alpha a1;
      beta b1;
      a1.display(b1);
}
```


Durr-e-Danish Library

Scanned by CamScanner

In this example the function `display()` which is a member function of class `alpha` is made a friend function of class `beta`. In the class specifier of class `beta` the function is declared as a friend. The class name of `alpha` is given with the scope resolution operator to identify the function. The friend function can access the private data member of class beta with the dot access operator.

The member functions of a class can all be made friends at the same time when you make the entire class a friend. For example,

Example 4

```
class beta;
class alpha{
  private:
      int data;
  public:
      friend class beta;          //beta is a friend class
};
class beta{


  public:
      void display(alpha d)       //can access alpha
      {cout<<d.data;}
      void get_data(alpha d)      //can access alpha
      { int x = d.data;}
};
```

By using the declaration,

```
friend class beta;
```

we make the class `beta's` entire member functions friends of `alpha`. Now all the member functions of class `beta` can access the private data members of `alpha`. However, the public member functions of the class `alpha` cannot access the private members of the class `beta`. It is also worth noting that friends of class `beta` are not automatically made friends of the class `alpha`. However, more than one class can be made a friend of class `alpha`.

3.3 Function Overloading

Other than classes and objects in object oriented programming, overloading of functions is an important feature. *Function overloading* is used to define a set of functions that are given the same name and perform basically the same operations, but use different argument lists.

Function overloading can be considered as a type of polymorphism called functional polymorphism. Polymorphism is essentially one thing having many forms. Therefore functional polymorphism means one function having several forms.

Durr-e-Danish Library
Scanned by CamScanner

It may seem mysterious how an overloaded function knows what to do. It performs one operation on one kind of data but another operation on a different kind.

For example,

```
void display();              // Display functions
void display(const char*);
void display(int one, int two);
void display(float number);
```

As far as the compiler is concerned, the only thing functions of the same name seem to have in common is the name. Seeing several functions with the same name but different number of arguments, the compiler could decide the programmer had made a mistake. Instead the compiler uses the context to determine which definition of an overloaded function is to be invoked. Which one of the functions will be called depends on the number and type of arguments supplied in the call. We can rely on the compiler to call the correct function.

➢ **Advantages**

The main advantages of using function overloading are:

- Eliminates the use of different function names for the same operation
- Helps to understand and debug code easily
- Maintaining code is easier

Function overloading is an exciting feature but it should not be overused. Only those functions that basically do the same task, on different sets of data, should be overloaded. Also, in function overloading, more than one function has to be actually defined and each of these occupy memory. In some cases, instead of function overloading, using default arguments may make more sense and create fewer overheads.

3.3.1 Overloading with various data types

The compiler can distinguish between overloaded functions with the same number of arguments, provided their type is different. By function overloading, the programmer's life is simplified a little since it reduces the number of function names to be remembered. Consider the following examples to find the square of a given number belonging to different data types:

```
int square(int);
float square(float);
double square(double);
```

You will notice that one method uses a single integer and another uses a single float type variable, but the system is able to select the correct one. You can use as many overloadings as desired provided all of the parameter patterns are unique.

You may think that this is a confusing thing to do but we have been using an overloaded function and you have not been the least confused over it. We have been using `cout`, which operates as an overloaded function since the way it outputs data depends on the type of its input variable or the field we ask it to display. We can use `cout` with an integer variable, a float variable, or any

Durr-e-Danish Library
Scanned by CamScanner

other data type and it displays the values as it is required. Many programming languages have overloaded output functions so that you can output any data with the same function name.

3.3.2 Overloading with different number of arguments

In addition to being overloaded for different data types, functions can also be overloaded for the number of arguments in the function call. Consider the following example:

```
int square(int);              //function declarations
int square(int,int,int);

int asq = square(a)           //function calls
int bsq = square(x,y,z)
```

When a function, `square`, is called, the compiler compares the types of the actual arguments with the types of the formal arguments of all functions called `square`. The idea is to invoke the function that is the best match on the arguments or else the compiler gives an error if no function produces the best match.

Note that the way the compiler resolves the overloading is independent of the order in which the functions are declared. In addition, the return types of the functions are not considered.

We have used constructors, which can be given different number of arguments or with different data types. This is an example of function overloading.

3.3.3 Scope rules for Function overloading

We know now that function overloading is the process of defining two or more functions with the same name, which differ only by the type of arguments and number of arguments. The overloading mechanism is acceptable only within the same scope of the function declaration. Sometimes, one can declare the same function name for different scopes of the classes or with global and local declaration, but it does not come under the technique of function overloading.

The following program segment shows how a function cannot be considered overloaded when it is declared for a different scope.

Example 5

```
class first{
      .
      .
      public:
            void display();
};
class second{
      .
      .
      public:
            void display();
};
```


Durr-e-Danish Library
Scanned by CamScanner

```
void main()
{
      first object1;
      second object2;
      object1.display();     //no function overloading takes place
      object2.display();
}
```

The same function `display()` is available in both functions. The scope is strictly confined to the classes in which they are declared.

3.4 Reference arguments

Arguments in functions can be passed by value or by reference. When arguments are passed by value, the called function creates a new variable of the same type as the argument and copies the argument's value into it. The function does not have access to the original variable in the calling program, only to the copy it created. Whatever changes are made to the copy does not affect the original variable. This is useful when the function does not have to modify the original variable in the calling program.

Passing arguments by reference uses a different technique. Instead of a value being passed to the function, a reference to the original variable in the calling program is passed

The main advantage of passing by reference is that the function can access the actual variable in the calling program. This method also provides a mechanism for returning more than one value from the function back to the calling program.

3.4.1 Passing references to functions

Pg. 181, 183 for S.

A *reference* provides an alias or an alternate name for an object. By far the most important use for references is in passing arguments to functions.

Let us look at an example, which will help clarify this mechanism:

Example 6

```
void swap(int& i, int& j)
{
  int tmp = i;
  i = j;
  j = tmp;
}
main()
{
  int x, y;
  .
  .
  swap(x,y);
}
```


Durr-e-Danish Library

Scanned by CamScanner

- When a function is supplied default values for arguments, calls to it need not include all the arguments shown in the declaration. The default values in the function are used for the missing arguments. Default values can be supplied for all the arguments or only for the trailing arguments.

- Friend functions can access a class's private data, even though they are not member functions of the class. This is useful when one function needs to have access to two or more unrelated classes.

- When all the members of a class need to be made friends you can make the entire class a friend.

- An overloaded function is a group of functions with the same name. Which of the function is executed when the function is called depends on the type of arguments and the number of arguments supplied in the call.

- The scope of an overloaded function is only within the same class as the function declaration.

- A reference provides an alias or an alternate name for an object. A call by reference does not result in passing a copy of the actual variable, only an alias is created, which can be used as a formal variable in the called function. Returning a reference does not return back a copy of the variable, instead an alias is returned.

- Inline functions look like normal functions in the source code but they insert the code directly into the calling program. Inline functions execute faster than normal functions but unless they are very small they may need more memory than normal functions.

Durr-e-Danish Library

Scanned by CamScanner

Check your Progress

1. A default argument in a function has a value that
   a. is a variable value
   b. is a constant value
   c. is supplied by the function
   d. increments each time the function is called

2. A friend function is used to avoid arguments between classes. ~~True/False~~

3. The keyword friend appears in
   a. the main() program
   b. the private or public section of a class
   c. the class allowing access to another class
   d. the class desiring access to another class

4. A friend class cannot be referred to unless it is first declared. True/~~False~~

5. Function overloading perform one operation on one kind of data but another operation on a different kind of data.

6. When an alias of a variable is passed to a function it is said to be a reference argument.

7. Inline functions are best reserved for small, frequently used functions.

8. Overloaded functions
   a. are a group of functions with the same name
   b. have the same number and types of arguments
   c. must have constant values for arguments
   d. save memory space

9. In general, an inline function executes faster than a normal function.

10. A function is made to return a reference by putting ampersand before its name.

Durr-e-Danish Library

Scanned by CamScanner

Do it Yourself

1. Write a declaration for a function called default_test() that takes two arguments and returns the type float. The first argument is type int and the second argument is type float with a default value of 2.217.

2. Write a function called small_test() that is passed two arguments of type int as reference. The function should find the smaller of the two numbers and change it to -1.

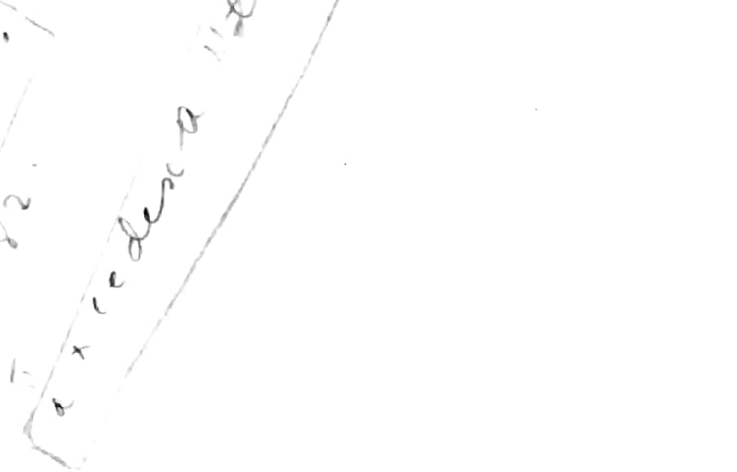
3. Write declarations for two overloaded functions named overboard(). The first function should take one argument of type char and the second should take two arguments of type char. Both functions return the type int.

4. Write the following:
   a. Declaration of a friend function called Barney() that returns type int and takes an object of class Fred as an argument.
   b. A class Fred with a declaration to make all members of the class Barney friends of the class.

5. Write the definition for an inline function named infun() that takes one argument of type int, squares it and returns type int.

6. Write a function, fn(), to increment the values of two variables of type int that are passed by reference.

7. Design a class date that can be used in many applications. You can make functions to store a given date, compare dates, print the date in different formats and add dates. Write a program to test this class.



Durr-e-Danish Library
Scanned by CamScanner

Session Objectives:

At the end of this session, the student will be able to -

- *Describe Operator Overloading*
  - *Unary operators*
  - *Binary operators*
    - *Binary arithmetic operators*
    - *Compound assignment operators*
    - *Comparison operators*
- *Describe overloading of the Assignment Operator*
- *Describe Copy Constructors*
- *Describe conversion functions which help in conversion*
  - *from Basic types to User-Defined types*
  - *from User-Defined types to Basic types*
  - *between Objects of different Classes*
- *Identify operators that cannot be overloaded*

4.1 Operator Overloading

Operator overloading is one of the interesting and useful features of object-oriented programming. In traditional programming languages, expressions involving operators like +, -, >, +=, ==, etc. can be used only on basic data types like `int` and `float` and not on derived or user-defined data types like objects. For example,

```
x = y + z;
  or
if(x>y) {. . .}
```

works only with basic types like `int` or `float`. If you tried to do the same operation on objects of a class the compiler would give an error. However, the concept of operator overloading allows statements like ,

```
if (obj1>obj2){ . . .}
```

where `obj1` and `obj2` are objects of a class. The operation of comparing the objects can be defined in a member function and associated with the comparison operator.

Definition

The ability to associate an existing operator with a member function and use it with objects of its class as its operands is called *operator overloading*.

Essentially, operator overloading gives you the ability to redefine the language by allowing you to change the way operators work. We have seen with overloaded functions that the compiler knows which function to use based on the data type of the arguments. In the same way, the

Durr-e-Danish Library

Scanned by CamScanner

compiler can distinguish between overloaded operators by examining the data type of its operators.

Operator overloading is one form of polymorphism. Polymorphism allows the creation of multiple definitions for operators and functions. In the previous session we have seen that overloading of functions is termed functional polymorphism. Operator overloading can be termed as *operational polymorphism.*

In C++, programs can overload existing operators with some other operations. If the operator is not used in the context as defined by the language then the overloaded operation, if defined, will be carried out. For example, in the statement

```
x = y-z;
```

if, x, y and z are integer variables, then the compiler knows the operation to be performed. If x, y and z are objects of a particular class, then the compiler will carry out the instructions that have been associated with the '-' operator.

Since the operator is being overloaded it is also possible to write instructions that would make the '-' operator function perform operations like addition or multiplication. However, that would not be contrary to the original specification for which the operator was intended.

There are certain points about overloading that are worth noting:

- Overloading cannot alter the basic function of an operator, nor change its place in the order of precedence that is already defined in the language. For example, ++ (increment) and -- (decrement) can be used only as unary operators.
- Overloading an operator should never change its natural meaning. For example, an overloaded + operator can be used to *multiply* two objects but this would make your code unreadable.
- Only operators that already exist in the language can be overloaded. You cannot use a new symbol. Barring a few exceptions, all the operators in C++ can be overloaded. The list of operators that cannot be overloaded will be given at the end of the session.
- Overloading of operators is only available for classes. You cannot redefine the operators for the predefined basic types.

The main advantage of using operator overloading is that it makes programs easier to read and debug. It is easier to understand that two objects are being added and the result assigned to a third object, if you use the syntax

```
obj3 = obj1 + obj2;
```

instead of the following statement,

```
obj3.addobjects(obj1,obj2);
```

When we use operator overloading in this manner, we actually make our programs look like the class is a natural part of the language since it is integrated so well into the language.

Durr-e-Danish Library
Scanned by CamScanner

➢ The operator keyword

The actual instructions to overload an operator are written in a special member function defined with the keyword `operator`. The operator that has to be overloaded follows the keyword. Such a member function is called an *operator function*. The general format of this function is

```
return_type operator op(argument list);
```

where op is the symbol for the operator that is being overloaded. For example in a class `Sample` the declaration,

```
void operator --();
```

tells the compiler to call this member function whenever the decrement operator is found in the program, provided the variable that is operated on by `--` is of the type `Sample`. For example,

```
Sample s1;
s1--;
```

calls the operator -- member function of the class Sample.

4.1.1 Unary Operators

Unary operators have only one operand. Examples of unary operators are the increment operator ++, the decrement operator --, and the unary minus operator.

The increment and decrement operators can be used as either prefix or postfix operations. Let us look at an example.

Example 1

```
class Sample{
  private:
      int counter;
  public:
      Sample()
      {counter =0;}
      void operator++()
      {++counter;}
};
void main()
{
  Sample obj1;
      obj1++;                  //increments counter to 1
      ++obj1;                  //increments counter to 2
}
```

In `main()` the increment operator is applied to a specific object. The member function does not take any arguments. It increments the invoking object's data member `counter`. A similar function to decrement the object can also be included in the class as:

Durr-e-Danish Library
Scanned by CamScanner

```
void operator --()
{--counter;}
```

This can be invoked with the statement

```
--obj1;
```
or
```
obj--;
```

In the above example, the compiler checks to see if the operator is overloaded. If an operator function is found in the class specifier, the statement to increment the object

```
obj1++;
```

gets converted, by the compiler to the following:

```
obj1.operator++();
```

This is just like a normal function call qualified by the object's name. The compiler treats it like any other member function of the class.

However, the operator function in the above example has a problem. On overloading, it does not work exactly as it does for basic data types. With the overloaded increment and decrement operators, the operator function is executed first, regardless of whether the operator is postfix or prefix.

Whether the operator is prefix or postfix will not matter in statements like `obj1++` or `++obj1` but there is a problem when the following statement is used:

```
obj1 = obj2++;
```

e.g (293).

If `obj1` and `obj2` were basic data types the compiler would not give any error. However, it will give an error if `obj1` and `obj2` are objects. The ++ operator function in the class is declared with a `void` return type, whereas we want the operator function to return a variable of type `Sample` so that it can be assigned to the object on the left-hand side. We will have to revise the function so that it is able to return an incremented object. To do that we can use a temporary object in the operator function as shown below:

Example 2

```
Sample Sample::void operator++()
{
  Sample temp;                  //create a temporary object
  temp.counter = ++counter;     //assign incremented value
  return temp;                  //return incremented object
}
```

In this example, the operator function creates an object `temp` of class `Sample`, assigns the incremented value of `counter` to the data member of `temp` and returns the new object. Now the object that is returned from the function can be assigned to another object in main. This is

Durr-e-Danish Library
Scanned by CamScanner

just one way of returning the object. Another way is to create a nameless temporary object and return it.

Example 3

```
class Sample{
  private:
      int counter;
  public:
      Sample()                    //constructor with no argument
      {counter = 0;}
      Sample(int c)               //constructor with one argument
      {counter = c;}
      Sample operator++();
};

Sample Sample::void operator++()
{
      ++counter;
      return Sample(counter); //or only return Sample(++counter);
}
```

One change in the class definition above is a constructor with one argument. No new temporary object is explicitly created in the operator function. However, in the return statement, an unnamed temporary object of class `Sample` is created by the constructor that takes one argument. The object is initialised with the value in `counter` and the function returns the object.

Yet another way of returning an object from the member function is by using the `this` pointer. In an earlier session we have seen that the `this` pointer is a special pointer that points to the object that invoked the member function. This is how we can use the `this` pointer to return an object.

Example 4

```
Sample Sample::void operator++()
{
      ++counter;
      return(*this);
}
```

The one argument constructor is not needed in this approach.

When ++ and -- are overloaded, there is no distinction between the prefix and postfix operation. The expressions,

```
obj2 = obj1++;
```
and
```
obj2 = ++obj1;
```


Durr-e-Danish Library

Scanned by CamScanner

produce the same effect. In both the cases `obj1` is incremented before it is assigned to `obj2`. This is different for basic data types where the first expression would cause assignment before incrementing. In this aspect the use of the ++ operator and -- operator with objects is not totally equivalent to its usage with the basic data types.

To overload the ++ operator for a postfix operation we will have to define the function with a single argument. For example,

Example 5

```
Sample Sample::void operator++(int)
{
      return Sample(counter++);
}
```

The `int` argument is never used. It is simply a dummy argument that allows the compiler to distinguish between the prefix and postfix operations. This function will have to be defined in the class along with the overloaded function that takes no argument.

4.1.2 Binary Operators

Binary operators are operators with two operands. They can also be overloaded just like unary operators. We will look at three different categories of binary operators

- Arithmetic operators
- Compound assignment operators
- Comparison operators

Binary operators can be overloaded in two ways:

1. as member functions they take one formal argument, which is the value to the right of the operator. For example, when the addition `obj1+obj2` has to be done the overloaded operator function for + is declared as,

   ```
   operator+(Sample obj2)
   ```

2. as friend functions they take two arguments. For example,

   ```
   operator+(Sample obj1,Sample obj2)
   ```

a. Binary Arithmetic Operators

Arithmetic operators are binary operators and therefore, require two operands to perform the operation.

Using the `Sample` class from Example 1 we can define an overloaded function for the operator + as given below:

Durr-e-Dáñish Library

Scanned by CamScanner

Example 6

```
Sample Sample::operator+(Sample a)
{
 Sample temp;                          //temporary object
 temp.counter = counter + a.counter;   //addition
 return temp;                          //return temp object
}
```

Now we are able to perform addition of objects with a statement,

```
obj3 = obj1 + obj2;                    //objects of class Sample
```

The operator + can access two objects. In the above statement, the object on the left side of the operator, `obj1`, is the one that will invoke the function and the object on the right hand side, `obj2`, is taken as the argument to the function call. In the operator function, the left operand (object `obj1`) is accessed directly since this is the object invoking the function. The right hand operand is accessed as the function's argument as `a.counter`.

Using the overloaded + operator it is also possible to perform multiple additions such as

```
obj4 = obj3 + obj2 + obj1;
```

where all the variables are objects of the class. This kind of multiple addition is possible only because the return type of the + operator function is an object of type `Sample`.

Languages like BASIC have an in-built facility to concatenate two character strings using the + operator. This facility is not available in C++ but we can overload a + operator to achieve the same effect.

For a class `String` that contains a data member, `str`, which is a character array of, say, 100 characters, we can define an operator function as shown below:

Example 7

```
String String::operator+(String ss)
{
  String temp;                         //make a temporary string

  strcpy(temp.str,str);                //copy this string to temp
  strcat(temp.str,ss.str);             //add the argument string
  return temp;                         //return temp string
}
```

The operator function concatenates the strings of the two objects using the standard string handling functions into the string of the temporary object and returns it. You can use the above function to add two strings as shown in the statements below:

```
String s1 = "Welcome";
String s2 = "to C++";
String s3;
```


Durr-e-Danish Library

Scanned by CamScanner

```
s3 = s1 + s2;
```

The above statement would produce the string `Welcome to C++`.

Similarly other binary arithmetic operators can also be overloaded so that you can subtract, multiply, and divide objects of the class.

b. Compound Assignment Operators

Compound assignment operators can also be overloaded like binary arithmetic operators. Operators such as the += operator combine assignment and addition in one step. Let us look at an example for an overloaded operator function using the same class `Sample`.

Example 8

```
void Sample::operator+=(Sample a)
{
 counter += a.counter;      //addition
}
```

In this example there is no need for a temporary object. The object, whose data member `counter` is changed, is the object that invokes the function. The function also needs no return value because the result of the assignment operator is not assigned to anything. The operator is used in expressions like

```
obj1 += obj2;
```

If you want to use this operator in more complex expressions such as

```
obj3 = obj1 += obj2;
```

then the function would need to have a return value. Then the member function declaration would be:

```
Sample Sample::operator+=(Sample a);
```

The return statement can be written as:

```
return Sample(counter);
```

where a nameless object is initialised to the same values as this object. Of course, you will have to define a constructor that takes one argument so as to be able to create the nameless object.

c. Comparison Operators

Comparison and logical operators are binary operators that need two objects to be compared. The comparison operators that can be overloaded include <, <=, >, >=, ==, and !=. If we use the following comparison with an overloaded > operator,

```
if (s2>s2)
```


Durr-e-Danish Library
Scanned by CamScanner

the overloaded operator accepts the object s2, on its right, as the argument to the function and the object s1, to its left, as the invoking object. Let us look at a typical example using the class String that will help us to overload a comparison operator.

Example 9

```
int String::operator>(String ss)
{
  return(strcmp(str,ss.str) > 0));
}
```

str of invoking object

str of object ss that is passed as argument

The return value of the comparison is an integer. The > operator is used to indicate whether the first string comes before or after the second in list that is alphabetically ordered. The same function can be adapted for use with the other comparison operators. You could also define overloaded comparison operators to compare string lengths if that is what is needed in your program.

4.2 Overloading the Assignment Operator

We have probably used the assignment operator = many times without going into the details of how it operates with objects. When we use a statement such as,

```
obj2 = obj1;
```

the default assignment operator simply copies the source object to the destination object byte by byte. Let us consider what would happen in a class where the data members contain pointers and have been allocated memory using the new operator.

Example 10

```
class String{
  private:
     char *str;
  public:
     String(char *s = "")                    //constructor
     {
     int length = strlen(s);
     str = new char[length+1];
     strcpy(str,s);
     }
     ~String(){delete[] str;}                //destructor
     void display(){cout<<str;}
  };

void main()
{
  String s1("Welcome to my world \n");
  String s2;
     s2 = s1;
```


Durr-e-Danish Library

Scanned by CamScanner

```
    s1.display();
    s2.display();
}
```

Two objects s1 and s2 are created. The constructor function allocates memory for a string and copies the contents of its formal argument in it. The assignment in main() assigns the object s1 to s2. The output of this program will be:

```
Welcome to my world
Welcome to my world
Null pointer assignment
```

We shall see why the program generates a null pointer assignment message.

After the assignment the data member str of both the objects points to the same location of memory. That is because, the assignment operator copies only the pointer and not the actual contents of the string. As a result there are two pointers to the same memory location. After the program is executed, the destructor function is called automatically. This releases the memory allocated by new to the data member str. The delete operator called for one object releases the memory and for the second call it attempts to release the same memory location all over again. This results in the error message.

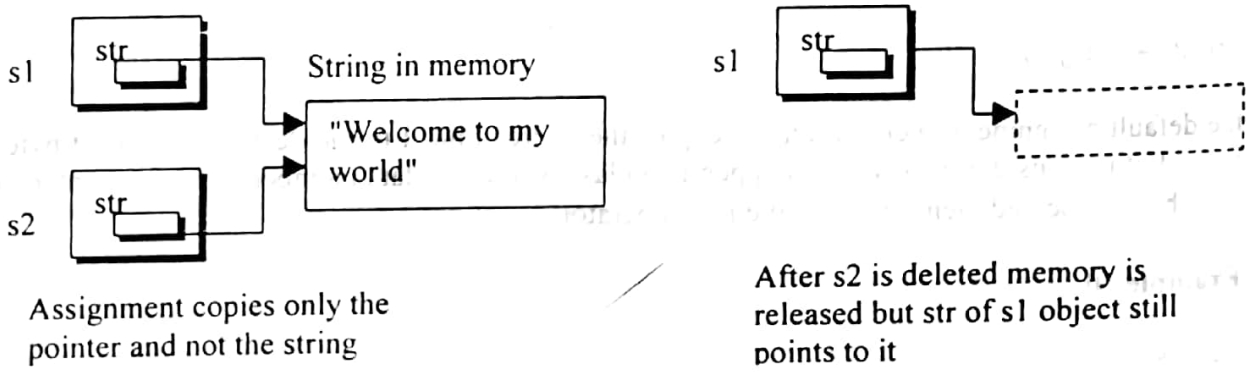


Figure 1: Assignment and deletion

The solution to this is to define an operator function for the assignment operator as given below:

```
String& String::operator=(String& s)
{
    delete str;
    int length = strlen(s.str);
    str = new char[length+1];
    strcpy(str,s.str);
    return(*this);
}
```

By including this function in the program the error message will not appear. When we use the assignment operator an existing object is being copied to another object. Since it is a member-by member-copy the pointer that is copied will be pointing to the old object. To avoid this, the old String object being pointed to by str is first deleted. A new space is allocated in memory for

Durr-e-Danish Library

Scanned by CamScanner

the string. The pointer `str` of the new object is made to point to this new space. The value of the old `String` object is copied into the new `String` object. Thus the problem of two pointers to the same location of memory is avoided.

Let us see an example of a similar function used for a data member that is an integer, as in class `Sample` used in Example 1.

```
Sample Sample::operator=(Sample& a)
{
      counter = a.counter;
      return Sample(counter);
}
```

When you overload the assignment operator you must make sure that all the data members are copied from one object to the other in the operator function. In the `Sample` class there is only one data member. Where more data members are involved each has to be copied to the target object.

You can see that the argument in the operator function is passed by reference. We have seen earlier that an argument passed by value creates a copy of itself in the function to which it is passed. This holds true for the `operator=()` function. If such objects are large each copy operation could take up a lot of memory. Values that are passed by reference do not create copies and hence save much needed memory space.

The operator function returns the value by creating a temporary `Sample` object and initialising it using a constructor that takes one argument. The value returned is a copy of the object of the same class that the member function belongs to. Thus, it is possible to use a chain of = operators, such as:

```
obj3 = obj2 = obj1;
```

Returning by value has the same problems of memory as passing an argument by value. Unfortunately we cannot return by reference any variables that are created as local variables in the function. Returning by reference returns only the address of the variable being returned. For local variables the address points to data within the function. The local variable is destroyed when the function is terminated and a pointer to it will have no meaning.

4.3 Copy Constructors

The problems that we have seen while using the assignment operator can also occur whenever we try to initialise an object with another object of the same class. Although initialisation and assignment both assign values, initialisation occurs only once when an object is created, whereas assignment can occur whenever it is wanted in the program. Let us take a look at two statements:

```
String s1("This is a string.");
String s2(s1);
```

The first statement declares an object `s1` and passes a string as an argument to its constructor. The second statement declared another object `s2` and contains an object as its argument. Since we have still not defined a constructor with an object as its formal argument, the compiler itself

Durr-e-Danish Library

Scanned by CamScanner

initialises the data members of `s2` with those of `s1`. Since the data member of class `String` is a pointer, the null pointer assignment problem arises just as in the assignment operator. The problem is solved by defining a constructor function that takes an object as its argument. This constructor is called the *copy constructor*. The general format for the copy constructor is:

```
X::X(X &ptr)
```

where `X` is the user defined class name and `ptr` is an object of class `X` that is passed by reference.

The copy constructor may also be used in the following format using the `const` keyword:

```
X::X(const X &ptr)
```

By using the const keyword we make sure that the copy process does not inadvertently make any changes to the object that is being copied.

We have seen that an object is copied when one object is used to initialise another object. There are two other instances when an object is copied. When an argument is passed to a function, a variable that has not been initialised so far, the formal argument, is initialised. This invokes the copy constructor. The same process of initialisation happens when a function returns an object. Thus we can see that a copy constructor is called in three contexts:

- when an object of a class is initialised to another of the same class
- when an object is passed as an argument to a function
- when a function returns an object.

The copy constructor is generated automatically for you by the compiler if you do not define one yourself. It is used to copy the contents of an object to a new object during construction of that new object. If the compiler generates it for you, it will simply copy the contents of the original into the new object as a byte by byte copy, which may not be what you want. For simple classes with no pointers, that is usually sufficient, but when, we have a pointer as a data member, a byte by byte copy would copy the pointer from one to the other and they would both be pointing to the same allocated member. A copy constructor for the `String` class is given below:

```
String::String(String& ss)
{
      int size = strlen(ss.str);
      str = new char[size+1];
      strcpy(str,ss.str);
}
```

This constructor contains an object as its argument. It allocates a new memory space for a string and `str` is made to point to this space. It then copies the contents of the string to this new location.

Assignment and initialisation are different operations. When a class X has a data member of pointer type, the class should have a constructor, assignment operator function, copy constructor and destructor. A typical class should be as given below:

Durr-e-Danish Library
Scanned by CamScanner

```
class X{
.
.
X(some_value);              //constructor
X(const X&);                //copy constructor
X& operator=(const X&);     //assignment
~X();                       //destructor
};
```

4.4 Conversion Functions

Conversion functions are member functions used to convert objects to or from basic data types and for conversions between objects of different classes. In the previous session we have seen how type conversion works for basic data types. However the compiler knows nothing about converting user defined types such as objects. The program has to define the conversion functions. Let us look at an example program, which we will use for the different conversions.

Example 11

```
class Converter
{
 private:
      int feet;
      float inches;
 public:
      Converter()                    //default constructor
      {feet = 0; inches =0.0;}
      Converter(float metres)        //constructor with one arg
      {
      float f;
      f= 3.28 * metres;
      feet = int(f);
      inches = 12 *(f-feet);
      }
};
void main()
{
      Converter d1 = 1.55;           //uses second constructor
      Converter d2;                  //uses first constructor
      d2 = 2.0;                      //uses second constructor
}
```

The class contains two data members, `feet` and `inches`. It stores a value, given in terms of metres, as feet and inches. The first constructor with no arguments initialises the data members to zero. The second constructor takes the argument `metres`, which is a `float` and converts it to feet and inches.

Durr-e-Danish Library

Scanned by CamScanner

4.4.1 From Basic types to User-Defined types

In Example 11, the declaration of the object d1 uses the second constructor and assigns the value 1.55. It shows the conversion of a float constant to an object of class Converter. The statement

```
d2 = 2.0;
```

also uses the constructor function for assignment of a float to object d2. The compiler first checks for an operator function for the assignment operator. If the assignment operator is not overloaded, then it uses the constructor to do the conversion. If the constructor also had not been defined the compiler would have given an error. We can see that the constructor function does not distinguish between initialisation at the time of declaration and assignment later on.

4.4.2 From User-Defined types to Basic types

Conversion of one basic data type to another is automatically done by the compiler using its own built-in routines or with the use of typecasting. Since the compiler knows nothing about a user-defined type such as an object, it has to be explicitly instructed if an object has to converted to a basic data type.

These instructions are coded in a conversion function and defined as a member of that class. In Example 11 above, we will have to add a member function, as given below, to convert an object of the class Converter to a float variable.

```
operator float()
{
 float f;
      f = inches/12;
      f += float(feet);
      return (f/3.28);
}
```

This conversion function can be used to convert an object of class Converter to a float variable implicitly using the statement,

```
m = d2;
```

or explicitly using statement,

```
m = float(d1);
```

This conversion function is nothing but overloading of the type cast operator. The conversion function contains the operator keyword and instated of an operator symbol it contains the data type. The compiler first checks for an operator function for the assignment operator and if that is not found it uses the conversion function. The conversion function must not define a return type nor should it have any arguments.

4.4.3 Conversion between Objects of Different Classes

Conversion of an object of one class to an object of another class can be done using the assignment operator, but since the compiler does not know anything about user-defined types.

Durr-e-Danish Library
Scanned by CamScanner

conversion functions have to be specified. This function can be a member function of the source class (i.e. the right hand side of the assignment operator) or it can be a member function of the destination class (i.e., the left-hand side of the assignment operator). In the following statement,

```
objectA = objectB;
```

objectA is considered an object of the destination class and objectB is an object of the source class.

Conversion of objects of two different classes can be achieved either with a one-argument constructor or a conversion function. Conversion functions are typically defined in the source class and one-argument constructors are typically defined in the destination class.

Let us look at two classes that can store a given length. The first class LFeet stores the length in terms of feet and inches and has a constructor to receive lengths in terms of feet and inches. The second class LMetres stores the length in metres and has a constructor to receive the length in terms of metres. Let us look at the class definitions first and then look at the conversion functions needed.

Example 12

```
class LFeet
{
 private:
      int feet;
      float inches;
 public:
      LFeet(){feet = 0; inches =0.0;}        //Constructor 1
      LFeet(int ft,float in)                 //Constructor 2
      {
      feet = ft;
      inches = in;
      }
};

class LMetres
{
 private:
      float metres;
 public:
      LMetres(){metres = 0.0;}               //Constructor 1
      LMetres(float m)                       //Constructor 2
      {
      metres = m;
      }

};
void main()
{
 LMetres dm1 = 1.0;
 LFeet df1;
```


Durr-e-Danish Library

Scanned by CamScanner

```
    df1 = dm1;
}
```

In `main()`, to be able to use the statement,

```
df1 = dm1;
```

i.e. to convert from one class to another we will have to define either a conversion function in the source class or a constructor function in the destination class. We will look at example of both.

➢ A Conversion Function in the Source Class

The conversion function in the source class to convert the length from the source class `LMetres` to the destination class `LFeet` would be as follows:

```
operator LFeet()                    //conversion function
{
 float ffeet, inc;
 int ifeet;
      ffeet = 3.28*metres;
      ifeet = int(ffeet);
      inc = 12*(ffeet - ifeet);
      return LFeet(ifeet,inc);
}
```

This conversion function will be defined in the source class `LMetres`. The statement to convert one object to another,

```
df1 = dm1;
```

calls the conversion function implicitly. It could also have been called explicitly as,

```
df1 = LFeet(dm1);
```

This statement shows that the conversion function is nothing but a member function for the overloaded cast operator. The use of this overloaded cast operator is similar to the one in section 4.4.2 above, where we converted an object to float.

➢ A Constructor Function in the Destination Class

The constructor function in the destination class should be as given below:

```
LFeet::LFeet(LMetres dm)            //constructor function
{
 float ffeet;
      ffeet = 3.28*dm.GetMetres();
      feet = int(ffeet);
      inches = 12*(ffeet - feet);
}
```


Durr-e-Danish Library
Scanned by CamScanner

This is defined in the class `LFeet`. A similar constructor function was used in section 4.4.1 above to convert a float to an object. The constructor function can also be used at the time of initialisation of an object. In addition, we will also have to define a member function called `GetMetres()` in the source class `LMetres` as given below:

```
float LMetres::GetMetres()
{
  return metres;
}
```

This function returns the data member metres of the invoking object. This function is required because the constructor is defined in the `LFeet` class and since `metres` is a private data member of the `LMetres` class, it cannot be accessed directly in the constructor function.

The use of a conversion function in the source class or a constructor function in the destination class is strictly a matter of choice. The following table summarises the conversion approaches we have seen.

Type of Conversion	Function in Destination Class	Function in Source Class
Basic to Class	Constructor	N/A
Class to Basic	N/A	Conversion Function
Class to Class	Constructor	Conversion Function

Table 1: Table for Type Conversions

4.5 Operators that cannot be overloaded

Not all operators can be overloaded. Here is a list of operators that cannot be overloaded:

- The sizeof() operator
- The dot operator (.)
- The scope resolution operator (::)
- The conditional operator (?:)
- The pointer-to-member operator (.*)

Durr-e-Danish Library

Scanned by CamScanner

The Session in Brief

- The ability to associate an existing operator with a member function and use it with objects of its class as its operands is called operator overloading. Operator overloading gives you the ability to redefine the language by allowing you to change the way operators work.
- Overloading cannot alter the basic function of an operator, nor change its place in the order of precedence that is already defined in the language.
- Overloading an operator should never change its natural meaning.
- Only operators that already exist in the language can be overloaded. You cannot use a new symbol.
- Overloading of operators is only available for classes. You cannot redefine the operators for the predefined basic types.
- The actual instructions to overload an operator are written in a special member function defined with the keyword `operator`. The operator that has to be overloaded follows the keyword. Such a member function is called an operator function.
- Unary operators have only one operand. Examples of unary operators are the increment operator ++, the decrement operator --, and the unary minus operator.
- With the overloaded increment and decrement operators, the operator function is executed first, regardless of whether the operator is postfix or prefix. Different operator functions have to be defined to distinguish between the postfix and prefix operations.
- Binary operators that are overloaded, take one formal argument, which is the value to the right of the operator. When binary operators are overloaded by means of friend functions they take two arguments.
- Binary operators that can be overloaded include binary arithmetic operators, compound assignment operators and comparison operators.
- Languages like BASIC have an in-built facility to concatenate two character strings using the + operator. This facility is not available in C++ but we can overload a + operator to achieve the same effect.
- Comparison and logical operators are binary operators that need two objects to be compared. The comparison operators that can be overloaded include <, <=, >, >=, ==, and !=.
- When you overload the assignment operator you must make sure that all the data members are copied from one object to the other in the operator function.
- The argument in the operator function for an assignment operator is passed by reference. This is useful when objects are large and each copy operation can take up a lot of memory. Values that are passed by reference do not create copies and hence save much needed memory space.
- A copy constructor is called in three contexts:

Durr-e-Danish Library
Scanned by CamScanner

- when an object of a class is initialised to another of the same class
- when an object is passed as an argument to a function
- when a function returns an object.

➢ The copy constructor is generated automatically for you by the compiler if you do not define one yourself. It is used to copy the contents of an object to a new object during construction of that new object.

➢ Although initialisation and assignment both assign values, initialisation occurs only once when an object is created, whereas assignment can occur whenever it is wanted in the program.

➢ Assignment and initialisation are different operations. When a class has a pointer variable, the class should have a constructor, assignment operator function, copy constructor and destructor.

➢ Conversion functions are member functions used to convert objects to or from basic data types and for conversions between objects of different classes.

➢ Conversion of an object of one class to an object of another class can be done using the assignment operator, but since the compiler does not know anything about user-defined types, conversion functions have to be specified. This function can be a member function of the source class or it can be a member function of the destination class.

➢ Conversion of objects of two different classes can be achieved with a one-argument constructor or a conversion function. Conversion functions are typically defined in the source class and one-argument constructors are typically defined in the destination class.

➢ Barring a few exceptions, all the operators in C++ can be overloaded. The operators that cannot be overloaded are the sizeof() operator, dot operator (.), scope resolution operator (::), conditional operator (?:), and pointer-to-member operator (.*)

Durr-e-Danish Library

Scanned by CamScanner

Check Your Progress

1. The meaning of an operator cannot be changed by overloading the operator.

2. The keyword Operator introduces an overloaded function definition.

3. Operator overloading allows us to create symbols for new operators ~~True~~/False

4. In a class X with three objects a1, a2, and a3, for the statement a3 = a1*a2; to work correctly, the overloaded * operator must
   a. take two arguments
   b. take no arguments
   c. use the object of which it is a member as an operand
   d. create a temporary object

5. In the definition of an overloaded unary operator we require No arguments.

6. When you overload a compound assignment operator, the result
   a. must be returned
   b. is assigned to the object on the right of the operator
   c. is assigned to the object on the left of the operator
   d. is assigned to the object of which the operator is a member

7. The operation of the assignment operator and the copy constructor are similar except that the copy constructor creates a new object. ~~True~~/False

8. A Copy constructor is invoked when a function returns a value.

9. The compiler does not automatically know how to convert between user-defined types and basic types. True/~~False~~

10. If there are two objects objectA and objectB which belong to different classes, the statement objectA = objectB will give an error. True/~~False~~

11. To convert from a user-defined type to a basic type, you would use a Conversion that is a member of the class.

Durr-e-Danish Library
Scanned by CamScanner

Do it Yourself

1. Write the complete definition for an overloaded ++ operator for the class `LFeet` from Example 12. It should add 1 to the feet member data so that you are able to execute the statement

   ```
   df++;
   ```

   where `df` is an object of the class `LFeet`.

2. Rewrite the operator function in Question 1 so that it is possible to execute statements such as

   ```
   df2 = df1++;
   ```

   where `df1` and `df2` are objects of class `LFeet`.

3. Write a program using a class called Alpha that has one data member of type int. The class should have constructors, an overloaded assignment operator and copy constructor. The program should display the result whenever an initialisation or assignment takes place. For example, display the result of executing the following statements:

   ```
   Alpha obj1(50);
   Alpha obj2;
   obj2 = obj1;
   Alpha obj3(obj1);
   Alpha obj4 = obj2;
   ```

4. Rewrite the program in Example 7 with overloaded >, == and < operators that work with the `String` class. The overloaded operators should allow you to check if a string s1 is longer than, shorter than or of the same length as another string s2.

5. Write a program with a class named Student. The class should have data members for the name and marks of a student. The program should check if a student's marks are above, below or equal to a pass mark of 35. If the student's marks are below 35 the student should be given an extra 5 marks. Use overloaded operator functions wherever possible. If the marks are still below 35 the student fails. Display the name and marks of the student and whether the student has passed or failed.

Durr-e-Danish Library

Scanned by CamScanner

This page has been intentionally left blank

Durr-e-Danish Library
Scanned by CamScanner

=) Something you get legally from your work in addition to your wages.

Chapter 5 Inheritance

Session Objectives:

At the end of this session, the student will be able to -

- *Describe Single Inheritance*
- *Describe Base classes and Derived class*
- *Access Base class members and use pointers in classes*
- *Describe types of inheritance*
- *Describe Constructors and Destructors under inheritance*
- *Describe how to call Member Functions of the Base Class and Derived Class*
- *Describe Container Classes*

5.1 Single Inheritance

Leading from the concept of classes in object-oriented programming, is another powerful feature called inheritance. In order to maintain and reuse class objects easily, we need to be able to relate classes of similar nature into another. For example let us consider a program in which we are dealing with people employed in an organisation. There are employees who perform different duties, like the director, a manager, a secretary, a clerk, etc. Now each of these employees has some common features that link all of them. For example all employees must have:

- name
- age
- employee id
- salary structure
- department

These are the common properties they share. In addition to this, a director has certain number of people working under him, added responsibilities, and probably a different set of perks also. If we were to group the employees in different levels they could be categorised as level 1 for the director, level 2 for a manager, and so on. If we needed to put all this information together to form a class in an object-oriented program we would first consider a common class called Employee. This class can then be subdivided into classes like Director, Manger, Secretary etc. Inheritance is the process of creating new classes from an existing class. Each of the subclasses is considered to be *derived* from the class Employee. The class Employee is called the *base* class and the newly created class is called the *derived* class. Earlier, we have referred to these classes as the superclass or parent class and subclass respectively.

=) base class is also known as super or parent class
& Derived class is also known as subclass

Durr-e-Danish Library

Scanned by CamScanner

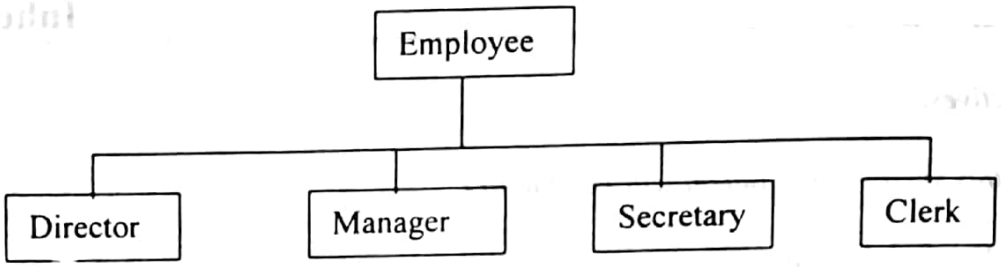


Figure 1: Class Employee and its subclasses

Definition

Single inheritance is the process of creating new classes from an existing base class.

The basic idea behind inheritance is that in a class hierarchy, the derived classes inherit the methods and variables of the base class. In addition they can have properties and methods of their own.

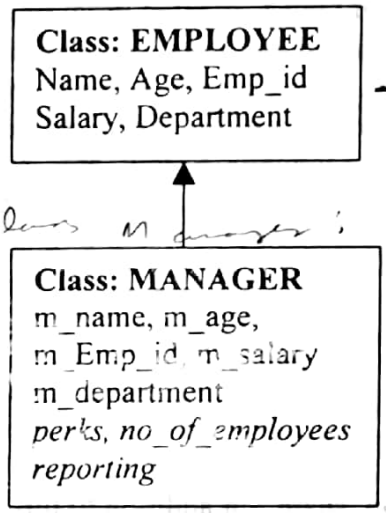


Figure 2: Derived class has properties of its own

The class Manager that is derived from the class Employee inherits the common properties of name, age, employee id, salary and department. In addition it has the data members perks and number of employees reporting, which are specific to the class Manager.

The most important advantage of inheritance is the reusability of code. Once a base class has been created it need not be changed but it can be adapted to work in different situations. One result of reusability of code is the development of class libraries.

Many vendors offer class libraries. A class library consists of data and methods encapsulated in a class. The source code of these class libraries need not be available to modify a class to suit one's needs. Modifying a class library does not require recompilation. Deriving a class from an existing one allows redefining a member function of the base class and also adding new members to the derived class. This is possible without having the source code of the class definition. All that is required is the class library, which does not require recompilation. The base class remains unchanged in the process.

Durr-e-Danish Library
Scanned by CamScanner

Derivation can be represented graphically with an arrow from the derived class to the base class as shown in Figure 3.

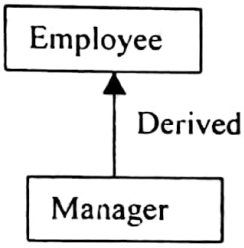


Figure 3: Manager derived from Employee

The arrow in the diagram is meant to show that the derived class is *derived from* the base class. The arrow pointing towards the base class signifies that the derived class refers to the functions and data in the base class, while the base class has no access to the derived class. These concepts will become clearer as we see them from the programming point of view.

The declaration of a singly derived class is similar to that of any ordinary class. In addition we also have to give the name of the base class. For example,

```
class Manager : public Employee
```

Any class can be used as a base class. This means that a derived class can in turn be the base for another class. Therefore, a base class can be classified into two types:

- direct base
- indirect base

A base class is called direct if it is mentioned in the base list. For example:

```
class A
{ };
class B : public A
{ };
```

where class A is a direct class. An indirect class can be written as:

```
class A
{ };
class B : public A
{ };
class C : public B
{ };
```

Here B is derived from A, and C from B. The class A is the indirect base class for C. This can be extended to an arbitrary number of levels.

We will see how the base class and derived class interact with each other.

Durr-e-Danish Library
Scanned by CamScanner

5.3 Accessing Base Class Members

An important part of inheritance is knowing when a member function or data member of a base class can be used by objects of the derived class. This is called *accessibility*. Let us first look at the base class members, which are defined with private and public access specifiers.

Class members can always be accessed by member functions within their own class, whether the members are private or public. Objects defined outside the class can access class members only if the members are public. For example, if `empl` is an instance of class `Employee`, and `display()` is a member function of `Employee`, then in `main()` the statement

```
empl.display();
```

is valid if `display()` is public. The object `empl` cannot access private members of the class `Employee`.

With inheritance, the member functions of the derived class can access members of the base class if its members are public. The private keyword makes a member of a class really private. Nothing but the member functions of the class itself can use it. The derived class members cannot access the private members of the base class.

Till now we have discussed only the public and private sections of a class. There is one more section called the protected section. The protected section is like the private section in terms of scope and access i.e., like private members, protected members can be accessed only by members of that class. Protected members cannot be accessed by objects or functions from outside the class, such as `main()`. This property of protected members is also similar to that of private members. However, the difference between them appears only in derived classes.

Private members of a class cannot be derived. Only public and protected members of a class can be derived. In other words, members of the derived class can access public and protected members; they cannot access the private members of the base class. This restriction is in conformance with the object-oriented concept of information hiding. The designer of a class may not want anyone to have access to some of the class members and those members can be put in the private section. On the other hand, the designer can allow controlled access by providing some protected members.

Inheritance does not work in reverse. The base class and its objects will not know anything about any classes derived from it. The following table summarises the access for the different sections of the base class:

Access specifier	Accessible from own class	Accessible from derived class	Accessible from objects outside the class
Public	Yes	Yes	Yes
Protected	Yes	Yes	No
Private	Yes	No	No

Table 1: Access rules for Base class members

Using the rules in the above table, let us look at an example.

Durr-e-Danish Library
Scanned by CamScanner

Example 1

```
class Employee{                                  //base class
  private:
      int privA;
  protected:
      int protA;
  public:
      int pubA;
};
class Manager : public Employee{                 //derived class
  public:
      void fn()
      {
       int a;
       a = privA;                                //error:not accessible
       a = protA;                                //valid
       a = pubA;                                 //valid
      }
};
void main()
{
  Employee emp;                                  //Object of base class type
       emp.privA = 1;                            //error:not accessible
       emp.protA = 1;                            //error:not accessible
       emp.pubA = 1;                             //valid
   Manager mgr;                                  //object of derived class
       mgr.privA = 1;                            //error:not accessible
       mgr.protA = 1;                            //error:not accessible
       mgr.pubA = 1;                             //valid

}
```

In the above example you can see that the function `fn()` can access the public and protected base class members from the derived class. However, in `main()` only the public base class member can be accessed.

While writing a class if you foresee it being used as a base class in the future, then any data or functions that the derived classes might need to access should be made protected rather than private.

➢ Pointers in classes

We need to understand how we can use a pointer, which has been declared to point to one class, to actually refer to another class. If we referred to an `Employee` we could be referring to a `Manager`, a `Secretary`, a `Clerk`, or any other kinds of `Employee`, because we are referring to a very general form of an object. If however, we were to refer to a `Manager`, we are excluding `Secretaries`, `Clerks`, and all other kinds of `Employees`, because we are referring to a `Manager` specifically. The more general term of `Employee` can therefore refer to many kinds of `Employees`, but the more specific term of `Manager` can only refer to a single kind of `Employee`, namely a `Manager`.

Durr-e-Danish Library

Scanned by CamScanner

We can apply the same idea and say that if we have a pointer to an `Employee`, we can use that pointer to refer to any of the more specific objects. Similarly, if we have a pointer to a `Manager`, we cannot use that pointer to reference any of the other classes including the `Employee` class because the pointer to the `Manager` class is too specific and restricted to be used on any of the other classes.

The general rule is that if a derived class has a public base class, then a pointer to the derived class can be assigned to a variable of type pointer to the base. For example, because a `Manager` is an `Employee`, a `Manager*` can be used as an `Employee*`. However, an `Employee*` cannot be used as a `Manager*`. For example, using the `Employee` and `Manager` classes of Example 1,

Example 2

```
void main()
{
  Manager mgr;
  Employee* emp = &mgr; //valid:every Manager is an Employee

  Employee eml;
  Manager* man = &eml;  //error: not every Employee is a Manager
}
```

Therefore, an object of a derived class can be treated as an object of its base class when manipulated through pointers. However, the opposite is not true. At a later stage we will see how this use of pointers will be a handy feature.

5.4 Types of Inheritance

C++ provides different ways to access class members. One of the ways for member access-control is the way that derived classes are declared. A derived class can be declared with one of the specifiers i.e., public, private and protected. Let us see what difference it makes when a derived class is declared with one of the three specifiers.

The keyword public in the class declaration of the derived class specifies that objects of the derived class are able to access public member functions of the base class. With the keyword private in the derived class declaration, objects of the derived class in `main()` cannot access public member functions of the base class. We have already seen that since objects outside the class can never access private or protected members of a class, no member of the base class is accessible to objects of the derived class. Since there are so many combinations for access let us first look at an example with publicly and privately derived classes to reinforce the concept.

Example 3

```
class A{                        //base class
  private:
      int privA;
  protected:
      int protA;
```


Durr-e-Danish Library
Scanned by CamScanner

```
  public:
      int pubA;
};

class B : public A              //publicly derived class
{
  public:
      void fn()
      {
       int a;
       a = privA;               //error:not accessible
       a = protA;               //valid
       a = pubA;                //valid
      }
};
class C : private A             //privately derived class
{
  public:
      void fn()
      {
        int a;
        a = privA;              //error:not accessible
        a = protA;              //valid
        a = pubA;               //valid
      }
};

void main()
{
  int m;
  B obj1;                       //object of publicly derived class
       m = obj1.privA;          //error:not accessible
       m = obj1.protA;          //error:not accessible
       m = obj1.pubA;           //valid: B is publicly derived from
                                // class A

  C obj2;                       //object of privately derived class
       m = obj2.privA;          //error:not accessible
       m = obj2.protA;          //error:not accessible
       m = obj2.pubA;           //error:not accessible: C is privately
                                // derived from class A
}
```

The example shows a base class, A, with private, protected, and public data members. The class B is derived publicly from A and C is privately derived.

If no access specifier is given while creating the class, private is assumed.

Functions in the derived classes can access protected and public members in the base class. You can observe this in the statements in the class declarations of class B and C. Objects of the derived classes outside the class or in `main()` cannot access private or protected members of the

Durr-e-Danish Library
Scanned by CamScanner

base class. This is what we have seen earlier. The difference is between *publicly* and *privately derived* classes.

Objects of the class B, which is *publicly* derived from A *can* access public members of the base class. However, objects of the class C, which is *privately* derived from A, *cannot* access any of the members of the base class. You can see this from the statements in `main()`.

If the derived class is derived using the keyword protected from the base class, the access control is similar to privately derived classes. In this case the functions in a derived class can access protected and public members of the base class. However, *objects* of the derived class (in main or outside the class) *cannot* access any of the members of the base class. The following table summarises the accessibility as it is called for the different access specifiers of the derived classes.

Base Class Members	Public Inheritance	Private Inheritance	Protected Inheritance
Public	Public	Private	Protected
Protected	Protected	Private	Protected
Private	Not inherited	Not inherited	Not inherited

Table 2: Accessibility for Derived classes

There is an easy way to remember this table. First of all, derived classes have no access to private members of a base class. Secondly, inheriting the base class publicly does not change the access levels of the members inherited by the derived class from the base. The other two access levels of the base class cause all inherited members to be of the same access level, as the base class (private for private base, protected for protected base).

The type of derivation we do i.e. either public, private or protected will affect the access the derived class functions have over the members of the base classes in a multi-level inheritance like the one shown in Figure 4.

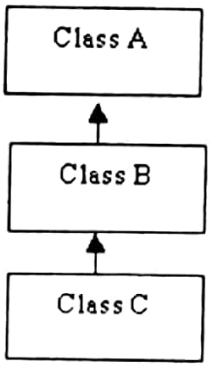


Figure 4: Type of Derivation in Multi-Level Inheritance

Example 4:

In the following code the `class B` derives privately from `class A` and `class C` in turn derives publicly from `class B`.

Durr-e-Danish Library
Scanned by CamScanner

```
class A
{
public :
      int a;
};
class B : private A
{
public :
      int b;
      void func_b()
      { int x,y;
         x=a;                          // valid
         y=b;                          // valid
      }
};
class C : public B
{
public :
      void func_c()
      {     int x,y;
                  x=a;                 // not valid
                  y=b;                 // valid
      }
};
```

In the above code, the `class B` is privately derived from `class A`. So, the data member **'a'** of `class A` can be accessed by the member functions of `class B` but becomes private after derivation. As a result the `class C` which even though being publicly derived from `class B` cannot access the data member **'a'**. Thus, in `class B` the function `func_b` can access the data member **'a'** while the `func_c` of `class C` cannot access it.

5.5 Constructors and Destructors under inheritance

The rules for constructing an object of a derived class are not complicated. The constructor of the base part of an object is first called, then the appropriate constructor of the derived class is called. For instance, consider a class `Derived` that derives from the class `Base`.

Example 5

```
class Base
{
protected:
      int a;
 public:
      Base(){a = 0;}                   //default constructor
      Base(int c){ a = c;}             //one-arg constructor
};
class Derived
{
public:
      Derived() : Base(){}             //default constructor
```


Durr-e-Danish Library
Scanned by CamScanner

```
        Derived(int c): Base(c){}   //constructor with one-arg
};
```

When you declare an object of the derived class, with the statement

```
Derived obj;
```

it will cause the constructor of the base class to be called first and then the constructor of the derived class. You can see that there is a difference in the declaration of the derived class constructor from constructors we have used so far. The base class constructor is given after the derived class constructor, separated by a colon, as in,

```
Derived(): Base(){}
```

This applies to not only the default constructors of the classes. You can explicitly select which constructor of the base class should be called during a call of the constructor of a derived class. The following statement

```
Derived obj1(20);
```

uses the one-argument constructor in Derived. This constructor also calls the corresponding constructor in the base class.

```
Derived(int c): Base(c);          //argument c is passed to Base
```

The argument `c` is passed from `Derived` to `Base`, where the one-argument constructor is used to initialise the object.

The derived class constructor is responsible for initialising the additional data members (if any) from the derived class, whereas the base class constructor is responsible for initialising the inherited members of the base class.

Destructors are called in the reverse order to constructors. This means that the destructors will be called for the derived class first, then for the base class. A destructor for the derived class is to be defined only if its constructor allocates any memory through dynamic memory managment. If the constructor of the derived class does not do anything or no additional data members are added in the derived class, then the destructor for the derived class can be an empty function.

5.6 Calling Member Functions

Member functions in a derived class can have the same name as those in the base class. When the function is invoked with the object of the base class, the function from the base class is called, whereas, when you use the name of the derived class object, the function from the derived class is invoked.

If a member function from the derived class wants to call the base class function of the same name, it must use the scope resolution operator as shown in the example below.

Durr-e-Danish Library
Scanned by CamScanner

Example 6

```
class Base
{
protected:
      int ss;
public:
      int func()
      {return ss;}
};
class Derived: public Base
{
public:
      int func()
      {return Base::func();}      //calling base class func
};
void main()
{
      Base b1;                    //base class object
      b1.func();                  //calls base class func
      Derived  a1;                //derived class object
      a1.func();                  //calls derived class func
}
```

Suppose, a function exists in the base class and not in the derived class. This function can be invoked using the objects of the base class as well as the derived class. But, if a function exists in the derived class and not in the base class, it can be invoked only with the objects of the derived class. The object of the base class does not know anything of the derived class and will always use only the base class functions.

5.7 Container classes

When a class X is publicly derived from another class Y it is said that X *is a* Y. In our example of the employee, the manager is an employee or the secretary is an employee etc. Thus, inheritance can be termed as an "is a" relationship. In contrast, a class X that has a member of another class Y is said to *have a* Y or X *contains* Y. This relationship is called a membership or a "has a" relationship.

```
class X{          //X contains Y
.
.
public:
      Y abc;
};
```

When a class contains an object of another class as its member it is called a *container class*. Consider the example of a jet plane. You might want to consider deriving a class for a jet plane from a class called engine. However a jet plane *is* not an engine but it *has* an engine. To be able to decide whether to use inheritance or membership you can ask whether a jet plane has more than one engine. If that is a possibility, it is most likely that the relationship will be "has a" rather

Durr-e-Danish Library

Scanned by CamScanner

than "is a". Not all relationships will fall under clear-cut categories. Therefore using a derived class or a container class is a matter to be considered at the design stage.

Let us look at an example of a container class to understand how constructors are defined.

Example 7

```
class engine:
{
 private:
      int num;
public:
      engine(int s)
      { num = s;}
};
class jet
{
 private:
      int jt;
      engine eobj;                    //declaring an object here
 public:
      jet(int  x, int y): eobj(y)
      { jt = x; }
};
```

As you can see, the class `jet` contains an object `eobj` in its private section. It also contains a constructor to which two variables `x` and `y` are passed. The variable `x` is used to initialise the private data member of the class `jet`.

In the constructor of class `jet` we would expect that after the colon a base class would be called. However, we are not dealing with inheritance and there is no base class. In this case, the name of the object of the class engine is written after the colon. It tells the compiler to initialise the `eobj` data member of class `jet` with the value in `y`. It is exactly like declaring an object of the class engine with the statement,

```
engine eobj(y);
```

Variables of any data type can be initialised like this.

Durr-e-Danish Library

Scanned by CamScanner

The Session in Brief

- Inheritance is the process of creating new classes from an existing class. Each of the subclasses is considered to be derived from the base class Employee.
- The base class and derived class can also be called the superclass or parent class and subclass respectively.
- Single inheritance is the process of creating new classes from an existing base class.
- The most important advantage of inheritance is the reusability of code. Once a base class has been created it need not be changed but it can be adapted to work in different situations. One result of reusability of code is the development of class libraries.
- The derived class can refer to the functions and data in the base class, while the base class has no access to the derived class.
- A derived class can in turn be the base for another class. This leads to the concept of direct base and indirect base classes.
- Private members of a class cannot be derived. Only public and protected members of a class can be derived. In other words, members of the derived class can access public and protected members only, they cannot access the private members of the base class.
- An object of a derived class can be treated as an object of its base class when manipulated through pointers. However, the opposite is not true.
- A derived class can be declared with one of the access specifiers i.e., public, private and protected.
- The keyword public in the class declaration of the derived class specifies that objects of the derived class are able to access public member functions of the base class.
- With the keyword private in the derived class declaration, objects of the derived class cannot access public member functions of the base class.
- Since objects can never access private or protected members of a class, no member of the base class is accessible to objects of the derived class.
- If the base class is declared as protected in the derived class, the access control is similar to privately derived classes. Functions in a protected derived class can access protected and public members in the base class.
- The constructor of the base part of an object is first called, then the appropriate constructor of the derived class is called. You can explicitly select which constructor of the base class should be called during a call of the constructor of a derived class.
- Destructors are called in the reverse order to constructors. Destructors will be called for the derived class first, then for the base class.

Durr-e-Danish Library

Scanned by CamScanner

- Member functions in a derived class can have the same name as those in the base class. When the function is invoked with the object of the base class, the function from the base class is called, whereas, when you use the name of the derived class object, the function from the derived class is invoked.

- If a member function from the derived class wants to call the base class function of the same name, it must use the scope resolution operator.

- When a class contains an object of another class as its member it is called a container class. This relationship is called a membership or a "has a" relationship.

Durr-e-Danish Library

Scanned by CamScanner

Check Your Progress

1. If the class Alpha inherits from the class Beta, class Alpha is called the derived class and class Beta is called the base class.

2. Inheritance enables reusability of code. which saves time in development and encourages using class libraries.

3. An object of a derived class can be treated as an object of its corresponding public base class.
True/~~False~~

4. A derived class can be made the base for another class. True/~~False~~

5. The three member access specifiers are private, protected and public.

6. A derived class cannot access the public members of the base class. ~~True~~/False

7. When deriving from a base class with protected inheritance, public members of the base class become public members of the derived class, and protected members of the base class become public members of the derived class.

8. A "has a" relationship between classes represents Containing class and an "is a: relationship between classes represents inheritance.

9. The private members of a class can be accessed by:
   a. Member functions of the class
   b. Non-member functions of the class
   c. Member functions of a derived class if it is declared privately
   d. Member functions of a derived class if it is declared publicly

10. Classify the following into "is a" and "has a" relationships

    a. country and capital -> is a. (inheritance).
    b. file and records -> has a
    c. modem and input/output devices -> has(relationship) is a
    d. car and door has a
    e. bicycle and vehicle -> has a (relation ship).

Durr-e-Danish Library

Scanned by CamScanner

Do it Yourself

1. Given the following classes:

```
class Alpha{
   public:
   int x;
};
class Beta: public Alpha
{
   public:
   int x;
};
```

   a. Write statements in a main() program that will allow you to assign a value to the data member of the base class and the derived class.

   b. If the data members of the base class and derived class were private, write statements to assign values to the data members in main().

   c. Using the above classes with public data members, write default constructors and one-argument constructors for both the classes.

2. Design a class of Shapes that can be subdivided into rectangles, triangles, circles, and ellipses. Use inheritance to classify the shapes, look for common features look for common features such as width, height, alignment point, and methods such as draw, initialise, set_alignment that can be part of the base class, and see if the shapes can be divided further into subclasses.

3. Design the class Employee that is referred to in this session with private data members name, employee id, designation and department, public member functions to get the information from the user and display it, Design a manager class with private data members for perks (you can give a figure in terms of dollars) and number of employees reporting to the manager. Also design a secretary class with an additional data member giving the name of the boss that the secretary reports to. Give suitable member functions to display these details. Write a program to test the classes and display all the information given by the user.

4. Create class Point. It has the data members x-co-ordinate and y-co-ordinate. The class declaration for Point should also contain member functions to set the point, get the x-co-ordinate and y-co-ordinate. It should have a constructor with default arguments. Create another class called Circle, which contains a data member called radius and an object of class Point, in which we can store the centre point of the circle. Note that this is a container class. The class Circle has a constructor with default arguments. It should have the facility to set the radius, get the radius and calculate the area. Also include a member which prints the details of the circle (i.e. centre point, radius, area).

❖ ❖ ❖ ❖ ❖

Durr-e-Danish Library
Scanned by CamScanner

Chapter 6 Multiple Inheritance and Polymorphism

Session Objectives:

At the end of this session, the student will be able to -

- *Describe Multiple Inheritance*
  - *Constructors under Multiple Inheritance*
  - *Ambiguity in Multiple Inheritance*
  - *Multiple Inheritance with a Common Base*
- *Describe Virtual Base Classes*
  - *Constructors and Destructors*
- *Use Pointers to Objects to access Member Functions*
- *Describe Virtual functions*
- *Describe Polymorphism*
- *Describe Dynamic binding*
- *Describe Pure Virtual Functions*
- *Describe Abstract classes*
- *Describe Virtual destructors*

6.1 Multiple Inheritance

A class can be derived from more than one class. This process is called multiple inheritance. As an example of multiple inheritance let us consider a program which models an educational institution, say, a college. In this institution we have teachers and students. Let us consider that the college employs some junior teaching assistants or research associates. It very often happens that to qualify for further promotions the teaching assistant has to acquire higher educational qualifications. In this case the teaching assistant falls under the category of a student also. Thus, the teaching assistant can derive properties from the class teacher as well as the class student. There are two base classes, the teacher and the student and the derived class is the teaching assistant.

Definition:

Multiple inheritance **is the process of creating a new class from more than one base class.**

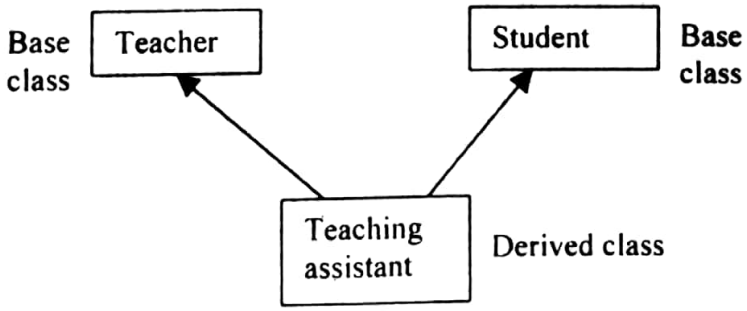


Figure 1: Multiple Inheritance

Durr-e-Danish Library
Scanned by CamScanner

The derived class inherits the properties of two or more base classes. Multiple inheritance can combine the behaviour of many base classes in a single class. In a single inheritance hierarchy, a derived class represents a specialised case of its base class. A multiple inheritance hierarchy represents a combination of its base classes.

The syntax for multiple inheritance is similar to that for single inheritance.

```
class Teacher
{ };
class Student
{ };
class Teach_asst: public Teacher, public Student
```

The names of both the base classes are provided separated by a comma. The rules of inheritance and access for multiple inheritance are the same as for single inheritance.

6.1.1 Constructors under Multiple Inheritance

The following example illustrates how constructors are handled in multiple inheritance.

Example 1

```
class Teacher
{
private:
      int x;
public:
      Teacher(){x =0;}                          //constructors
      Teacher(int s){x = s;}
};
class Student
{
private:
      int y;
public:
      Student(){y = 0;}                         //constructor
      Student(int a){y = a;}
};
class Teach_asst: public Teacher,public Student
{
private:
      int z;
public:
      Teach_asst():Teacher(),Student()          //constructor
      {z = 0;}
      Teach_asst(int s, int a, int b):Teacher(s),Student(a)
      {z = b;}
};
```


Durr-e-Danish Library
Scanned by CamScanner

Like with normal inheritance, constructors have to be defined to initialise the data members of all classes. The constructor in `Teach_asst` calls constructors for the base classes. The names of the base class constructor follow the colon and are separated by commas as in the statement,

```
Teach_asst():Teacher(),Student();          //constructor
```

If the constructors had arguments then the constructor in class `Teach_asst` would have to supply value for their arguments, besides arguments for its own constructor. We can see how this is done in the following statement.

```
Teach_asst(int s,                    arg for Teacher class
           int a,                    arg for Student class
           int b):                   arg for this class
           Teacher(s),               call Teacher constructor
           Student(a)                call Student constructor
      {z = b;}                       set own data member
```

The first two arguments passed to the `Teach_asst` constructor are passed on to `Teacher()` and `Student()`. The last argument is used to initialise the data member in the `Teach_asst` class.

When constructors are used in the base class and derived classes, the base class is initialised before the derived class, using either the default constructor or a constructor with arguments depending on the code of the constructor of the derived class. The base classes are initialised first, in the order they appear in the list of base classes in the declaration of the derived class. If there are member objects in the class, they are initialised next, in the order they appear in the derived class declaration (Example 1 does not have any member objects in the derived class). Finally the code of the constructor is called. For example in an object of class `Teach_asst`, the sequence of calls will be:

1. Base classes as they appear in the list of base classes: `Teacher, Student`
2. The object itself (using the code in its constructor)

The call for the destructor of a derived class follows the same rules as constructors, but in the reverse order. The destructor of the class is called first, then those of member objects, and then the base classes.

6.1.2 Ambiguity in Multiple Inheritance

Some problems can surface in cases where two base classes have the same function or data member name. The compiler will not be able to understand which function to use. Let us look at an example.

Example 2

```
class Alpha{
  public:
     void display();
};
class Beta{
```


Durr-e-Danish Library
Scanned by CamScanner

```
  public:
      void display()
};
class Gamma: public Alpha,public Beta
{ };
void main()
{
  Gamma obj;
       obj.display();       //ambiguous: cannot be compiled
}
```

Since there is an ambiguity about which `display()` function is called by the object `obj` the compiler will issue an error message. To access the correct function or data member you will need to use the scope resolution operator. For example,

```
obj.Alpha::display();
obj.Beta::display();
```

It is upto the programmer to avoid such conflicts and ambiguities. It can be resolved by defining a new function `display()` in the derived class `Gamma`, such as:

```
void Gamma::display()
{
   Alpha::display();
   Beta::display();
}
```

This localises the information about the base classes. Since `Gamma::display()` overrides the `display()` functions from both its base classes, this ensures that `Gamma::display()` is called wherever `display()` is called for an object of type `Gamma`. The compiler detects the name clashes and resolves it.

6.1.3 Multiple Inheritance with a Common Base

We have seen that it is possible to use more than one base class to derive a class. There are many combinations which inheritance can be put to. There is the possibility of having a class as a base twice. For example, in the above example of Teaching assistant, the classes Teacher and Student could have been derived from a class Person. Now Teaching assistant has a common ancestor - the Person class.

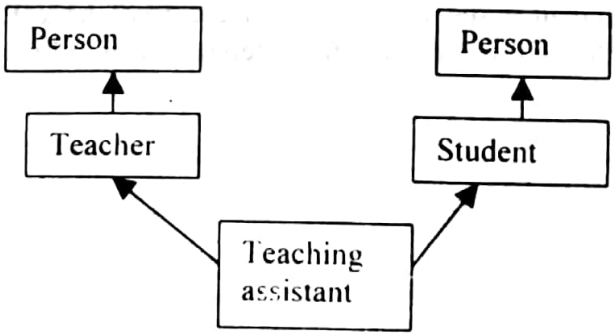


Figure 2: Multiple inheritance with common base

Durr-e-Danish Library
Scanned by CamScanner

One potential problem here is that both Teacher and Student contain a copy of the Person class members. Now when Teaching assistant is derived from both Teacher and Student it contains a copy of the data members of both classes. That means that it contains two copies of the Person class members - one from Teacher and one from Student. This gives rise to ambiguity between the base class data members. Another problem is that declaring an object of class Teaching assistant will invoke the Person class constructor twice. The solution to these problems is a virtual base class.

6.2 Virtual Base Classes

We have seen that multiple inheritance is the process of creating a new class from more than one base class. Multiple inheritance hierarchies can be complex, and may lead to a situation in which a derived class inherits multiple times from the same indirect class as we have seen above. For example, consider a class `window`, which has two directly derived classes `border` and `menu`. Now if we had another class `border_and_menu`, which is derived from the class `border` and `menu`, the class `window` would be a common base class.

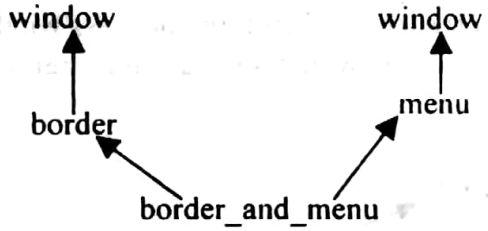


Figure 3 : Object with a common base class

Consider that the base class *window* has a data member `basedata`.

Example 3

```
class window{
  protected:
      int basedata;

};
class border: public window
{ };
class menu: public window
{ };
class border_and_menu: public border, public menu
{
 public:
      int show()
      {return basedata;}
};
```

A compiler error will occur when the `show()` function in the derived class `border_and_menu` tries to access `basedata` in the class `window`. When the classes `menu` and `border` are derived from `window`, each inherits a copy of `window`. This copy is called a

Durr-e-Danish Library
Scanned by CamScanner

subobject. Each of these subobjects contains its own copy of basedata from the class window. When the class border_and_menu refers to basedata the compiler does not know which copy is being accessed and hence the error occurs. To avoid two copies of the base class we use a virtual base class. To make a base class virtual just include the keyword *virtual* when listing the base class in the derived class declaration.

```
class border: virtual public window
{

};
```

and

```
class menu: virtual public window
{

};
```

The use of the keyword virtual in these two classes border and menu causes them to share a single common object of their base class window. Therefore, the class window is the *virtual base* . An object of border_and_menu can now be shown as given below:

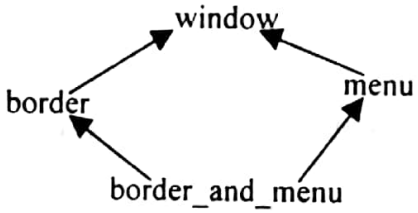


Figure 4 : Object with a virtual base class

This is different from multiple inheritance from a common base. Here, the virtual base class is represented by a single object of the class.

Virtual base classes are useful to avoid unnecessary repetitions of the same data member in a multiple inheritance hierarchy.

6.2.1 Constructors and Destructors

The presence of virtual base classes changes the order in which constructors are called. A virtual base class is initialised before any non-virtual base class. If there is more than one virtual base class, the order of initialisation is determined by their position in the inheritance graph from top to bottom and left to right.

The call for the destructor follows the same rules, but in the reverse order. The destructor of the class is called first, then those of its member objects, then the base classes and eventually the

Durr-e-Danish Library

Scanned by CamScanner

6.3 Pointers to Objects

Pointers can point to objects as well as to simple data types. We have seen how to use pointers in classes in the previous session. Consider a pointer to a class `Base` declared in a `main()` program.

```
Base* ptr;
```

We can use the `new` operator to create an object of the class and assign it to the pointer `ptr`.

```
ptr = new Base;
```

If we need to refer to the member functions in the object pointed to by `ptr` we cannot use a dot operator as we normally do with objects. The dot operator requires the identifier on its left to be a variable. Since `ptr` is a pointer we need to use the arrow operator (->) to access a member function, as in,

```
ptr->show();
```

The arrow operator works with pointer to objects in the same way that the dot operator works with objects. With inheritance, pointers can sometimes create problems when we need to access functions.

6.4 Virtual functions

The word virtual refers to something that exists in effect but not in reality. A virtual function is a function that does not really exist but does affect some parts of a program. Let us see why we would need to use virtual functions.

Consider a program to draw different shapes like triangles, circles, squares, ellipses etc. Consider that each of these classes has a member function `draw()` by which the object is drawn. To be able to put all these objects together in one picture we would have to find a convenient way to call each object's `draw()` function. Let us look at the class declarations for a class `Shapes` and derived classes `circle` and `square`.

Example 4

```
class Shapes
{
.
.
.
public:
      void draw()                    //function in the base class
      {cout<<"Draw Base\n";}
};
class circle: public Shapes
{
  private:
      int radius;
```


Durr-e-Danish Library

Scanned by CamScanner

```
  public:
      circle(int r);
      void draw()                    //redefined in derived class
      {cout<<"Draw circle";}
};
class square: public Shapes
{
  private:
      int length;
  public:
      square(int l);
      void draw()                    //redefined in derived class
      {cout<<"Draw square";}
};
void main()
{
  circle c;
  square s;
  Shapes* ptr;

  ptr = &c;
  ptr->draw();

  ptr = &s;
  ptr->draw();
}
```

When pointers are used to access classes the arrow operator (->) can be used to access members of the class. We have assigned the address of the derived class to a pointer of the base class as seen in the statement,

```
ptr = &c;        //c is an object of class circle
```

Now when we call the `draw()` function using,

```
ptr->draw();
```

we expect that the `draw()` function of the derived class would be called. However, the result of the program would be,

```
Draw Base
Draw Base
```

Instead, we want to be able to use the function call to draw a square or any other object depending on which object `ptr` pointed to. That means, different `draw()` functions should be executed by the same function call.

For us to achieve this all the different classes of shapes must be derived from a single base class Shapes. The `draw()` function must be declared to be `virtual` in the base class. The `draw()` function can then be redefined in each derived class. The `draw()` function in the base class declaration would be changed as shown below,

Durr-e-Danish Library

Scanned by CamScanner

```
class Shapes
{
.
.
.
public:
      virtual void draw()                    //virtual in the base class
      {cout<<"Draw Base\n";}
};
```

The keyword `virtual` indicates that the function `draw()` can have different versions for different derived classes. A *virtual function* allows derived classes to replace the implementation provided by the base class. The compiler makes sure the replacement is always called whenever the object in question is actually of the derived class. With the virtual function, if we were to execute the program in Example 4 we would get,

```
Draw circle
Draw square
```

It is the member functions of the derived classes that are executed and not that of the base class. The same function call,

```
ptr->draw();
```

executes different functions depending on the contents of `ptr`.

The virtual function should be declared in the base class and cannot be redeclared in a derived class. The return type of a member function must follow the keyword `virtual`. If a hierarchy of derived classes is used, the virtual function has to be declared in the top-most level. Here, the top-most class is `Shapes` and the virtual function is declared in it.

A virtual function must be defined for the class in which it was first declared (unless it is a pure virtual function as we shall see later). Then even if no class is derived from that class, the virtual function can be used by the class. The derived class that does not need a special version of a virtual function need not provide one.

To redefine a virtual function in the derived class you need to provide a function exactly matching the virtual function. It must have the same parameters (same number and data type), otherwise the compiler thinks you want to *overload* the virtual function. The return type does not have to match. For example, if the virtual function of a class returns a pointer to its class, you can redefine the function in a derived class and have it return a pointer to the derived class, instead of a pointer to the base class. This is possible only with virtual member functions.

The derived class can either fully replace ("override") the base class member function, or the derived class can partially replace ("augment") the base class member function. The latter is accomplished by having the derived class member function call the base class member function, if desired.

Durr-e-Danish Library

Scanned by CamScanner

6.5 Polymorphism

The word "poly" originates from a Greek word meaning "many" and "morphism" from the word meaning "form". Thus we have "polymorphism" which means "many forms". In more practical terms, polymorphism allows you to refer to objects of different classes by means of the same program item like a base class, and to perform the same operation in different ways depending on which object is being referred to at that moment.

For example, when describing the class mammals, you might note that eating is a fundamental "operation" for mammals to live. Every kind of mammal needs to perform the function EAT. However different mammals have different responses to the function EAT. A cow might ruminate in a field of grass, a lion might devour a deer, and a child might eat a chocolate fudge ice- cream. Polymorphism is the process of taking an object of type mammal and telling it to EAT. The object will perform the action that is most appropriate for it. Objects are polymorphic if they have some similarities but are still somewhat different.

Definition

Polymorphism **is the process of defining a number of objects of different classes in a group and using different function calls to carry out the operations of the objects.**

In other words, polymorphism means, "to carry out different processing steps by functions having the same messages".

The facility to invoke an appropriate function from any of the given classes using the pointer of the base class is a form of polymorphism. This is what we have done by using virtual functions.

We have earlier used the keyword virtual to define a base class. However, that is in a different context and is not related to polymorphism.

6.6 Dynamic binding

In earlier sessions we have seen polymorphism with reference to function overloading and operator overloading. Choosing a function in the normal way during compile time is called early binding or static binding. Non-virtual member functions are resolved statically. That is, the member function is selected statically (at compile-time) based on the type of the pointer (or reference) to the object. The compiler uses the static type of the pointer to determine whether the member function invocation is legal.

In contrast, virtual member functions are resolved dynamically (at run-time). That is, the member function is selected dynamically (at run-time) based on the type of the object, not the type of the pointer/reference to that object. This is called *dynamic binding* or *late binding*.

Definition

Dynamic binding **means that the address of the code in a member function invocation is determined at the last possible moment, based on the dynamic type of the object at run time.**

It is called dynamic binding because the binding to the code that actually gets called is accomplished dynamically (at run time). Dynamic binding requires some overhead in processing

Durr-e-Danish Library
Scanned by CamScanner

but provides increased power and flexibility in programming. Dynamic binding is a result of virtual functions.

If the keyword `virtual` is used with the function declaration in the base class, the system will use dynamic binding, which is done at run time, but if the keyword is not included, static binding will be used. What these words actually mean is that with dynamic binding, the compiler does not know which method will actually respond to the message because the object that a pointer points is not known at compile time. With static binding, however, the compiler decides at compile time what method will respond to the message sent to the pointer to an object.

6.7 Pure Virtual functions

Some classes such as class Shapes, represent abstract concepts for which objects cannot exist. A Shape makes senses only when it is the base of a class derived from it. It is not possible to provide concrete definitions for its virtual functions that will actually create a Shape object. An alternative is to declare the virtual function of the class Shape as a pure virtual function. A *pure virtual function* is a type of function, which has only a function declaration. A pure virtual function is declared by assigning the value of zero to the function as in,

```
virtual void getdata() = 0;
```

Every derived class must include a function definition for each pure virtual function that is inherited from the base class if it has to be used to create an object. This assures that there will be a function available for each call.

No function call will ever need to be answered by the base class since it does not have an implementation for a pure virtual function. You cannot create an object of any class, which contains one or more pure virtual functions, because there is nothing to answer if a function call is sent to the pure virtual method.

6.8 Abstract Classes

A class containing one or more pure virtual functions cannot be used to define an object. The class is therefore only useful as a base class to be inherited into a useable derived class. It is sometimes called an *abstract class*. No objects of an abstract class can be created. The abstract class can only be used as a base for another class. For example:

Example 5

```
class Shapes{
      .
      .
  public:
      virtual void draw() = 0;          //pure virtual function
      virtual void rotate(int) = 0;     //pure virtual function
};
class circle: public Shapes{
  private:
      int radius;
  public:
      circle(int r);
```


Durr-e-Danish Library
Scanned by CamScanner

```
        void draw();
        void rotate(int){ }
};
```

If a class inherits an abstract class without providing a definition for the pure virtual function, then it too becomes an abstract class and cannot be used to define an object.

An important use of abstract classes is to provide an interface without exposing any implementation details. You will find abstract classes used in many commercially available libraries and in application frameworks.

6.9 Virtual destructors

A destructor is invoked to free memory storage automatically. However the destructor of the derived class is not invoked to free the memory storage which was allocated by the constructor of the derived class. This is because destructors are non-virtual and the message will not reach the destructors under dynamic binding. It is better to have a virtual destructor function to free memory space effectively under dynamic binding. Let us look at an example.

Example 6

```
class Alpha{
private:
      char* alpha_ptr;
public:
      Alpha()                          //constructor cannot be virtual
      {alpha_ptr = new char[5];}
      virtual void fn();               //virtual member function
      virtual ~Alpha()                 //virtual destructor
      {delete[] alpha_ptr;}
};

class Beta: public Alpha{
private:
      char* ptrderived;
public:
      Beta()
      {ptrderived = new char[100];}
      ~Beta()
      {delete[] ptrderived;}
};
void main()
{
 Alpha *ptr = new Beta;
 delete ptr;
}
```

When you delete a derived object via a base pointer a virtual destructor is required. For example, when we use.

```
delete ptr;
```


Durr-e-Danish Library
Scanned by CamScanner

we are deleting a derived object using a pointer of the base class.

Virtual functions bind to the code associated with the class of the object, rather than with the class of the pointer/reference. When you say `delete ptr`, and the base class has a `virtual` destructor, the destructor that gets invoked is the one associated with the type of the object `*ptr`, rather than the one associated with the type of the pointer. As a derived class instance always contains a base class instance, it is necessary to invoke destructors of both the classes in order to ensure that all the space on the heap is released.

In general, it is a good idea to supply a `virtual` destructor in all classes that act as a base class. The idea behind this is, if you have any `virtual` functions at all, you are probably going to be doing some operations on derived objects via a base pointer, and some of the operations you may do may include invoking a destructor (normally done implicitly via `delete`).

A constructor cannot be virtual because it needs information about the exact type of the object it is creating to be able to construct it correctly.

Durr-e-Danish Library

Scanned by CamScanner

The Session in Brief

- Multiple inheritance is the process of creating a new class from more than one base class. The derived class inherits the properties of two or more base classes. Multiple inheritance can combine the behaviour of many base classes in a single class.

- The rules of inheritance and access for multiple inheritance are the same as for single inheritance.

- To avoid problems that can surface when two base classes have the same member name you can use a scope resolution operator.

- While using multiple inheritance with a common base there can be multiple repetitions of the base class members in the derived class. This can be avoided by using a virtual base class. Any base class that is declared using the keyword virtual is called a virtual base class.

- If we need to access the member functions in the object pointed to by a pointer we need to use the arrow operator (->).

- A virtual function is a function that does not really exist but does affect some parts of a program. A virtual function allows derived classes to replace the implementation provided by the base class. The compiler makes sure the replacement is always called whenever the object in question is actually of the derived class.

- The virtual function is declared in the base class and cannot be redeclared in a derived class. The return type of a member function must follow the keyword virtual. If a hierarchy of derived classes is used, the virtual function has to be declared in the top-most level.

- To redefine a virtual function in the derived class you need to provide a function exactly matching the virtual function.

- Polymorphism is the process of defining a number of objects of different classes in a group and using different function calls to carry out the operations of the objects.

- Virtual member functions are resolved dynamically (at run-time). That is, the member function is selected dynamically (at run-time) based on the type of the object, not the type of the pointer/reference to that object. This is called dynamic binding or late binding.

- Pure virtual functions are used in a base class that represents an abstract concept. A pure virtual function is a type of function, which has only a function declaration. Every derived class must include a function definition for each pure virtual function that is inherited from the base class if it has to be used to create an object.

- A class containing one or more pure virtual functions is called an abstract class. No objects of an abstract class can be created. The abstract class can only be used as a base for another class.

- When you delete a derived object via a base pointer a virtual destructor is required. In general, it is a good idea to supply a virtual destructor in all classes that act as a base class.

Durr-e-Danish Library

Scanned by CamScanner

Check Your Progress

1. C++ provides for Multiple Inheritance which allows a derived class to inherit from many base classes even if these base classes are not related.

2. A pure virtual function is specified by placing Zero at the end of its prototype in the class definition.

3. If a class contains one or more pure virtual functions it is called an abstract classes.

4. Polymorphism is implemented via virtual base classes. ~~True~~/False

5. If there is a pointer `ptr` to objects of a base class, and it contains the address of an object of a derived class, and both classes contain a non-virtual member function `getdata()`, then the statement `ptr->getdata();` will call the function of the base class.

6. A function call resolved at run time is referred to as dynamic binding and a function call resolved at compile time is referred to as static early binding.

7. There are many situations in which it is useful to define classes for which the programmer never needs to instantiate any objects. Such classes are:
   a. Virtual base classes
   b. Abstract classes
   c. Base classes
   d. Virtual abstract classes

8. A pointer to a base class can point to objects of a derived class. True/~~False~~

9. When you delete a derived object via a base pointer you require.
   a. Virtual constructor
   b. Pure virtual function
   c. Virtual destructor
   d. Virtual destructor and virtual constructor

Durr-e-Danish Library
Scanned by CamScanner

Do it Yourself

1. Create a class called base1 which has one data member called "value" which is of type integer. Create another class called base2, which has one data member called "index" which is of type integer. Both these classes have a constructor with default argument and another member function to get the data and both have the same name which is getdata(). Derive a class from the two above classes and name it as derived. It has one data member, which is called "real" and is of type float. The derived class has a constructor with default arguments and another function to get the value for "real". Test this class with a program and see to it that a derived class object calls the getdata() function.

2. Create a class called employee. It stores the first name and the last name. This information is common to all employees including derived classes. From class employee now derive classes hourlyworker, pieceworker, boss, commissionworker. The hourlyworker gets paid by the hour with 50% extra for overtime hours in excess of 40 hours per week. The pieceworker gets paid a fixed rate per item produced. Assume that this person makes only one type of item. The boss gets a fixed rate per week. The commission worker gets a small fixed weekly base salary plus a fixed percentage of that person's gross sales for the week. Every class should have a constructor, destructor and a function for displaying the data called print().

   Test the classes using the following specifications

   a. Do not use pointers and see to it that an object of every class calls its corresponding print() function.

   b. Use pointers. Assign a derived class object to a base class pointer and a base class object to a derived class pointer. Now call the print function and observe what happens.

   c. Make employee an abstract base class.

Durr-e-Danish Library

Scanned by CamScanner

Session 1

Session Objectives :

At the end of this session, the student will be able to -

- *Invoke Borland C++*
- *Write a new program*
- *Save the program*
- *Build the program*
- *Close the window*
- *Open an existing program*
- *Write programs to understand the concepts of*
  - *Classes*
  - *Objects*
  - *Member functions*
  - *Accessing, displaying and manipulating public data members*
- *Exit Borland C++*

The steps given in the session are detailed, comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

The First 1 Hour:

1.1 Invoking Borland C++

Borland C++ provides an almost ideal platform for learning C++. The Integrated Development Environment (IDE) puts all the tools you need for C++ program development into a single and convenient screen display. To invoke Borland C++ follow these steps:

1. **Click on the Start button.**

2. **Click on the option 'Shut Down...'.**
   This displays the dialog box, 'Shut Down Windows'. Refer Figure 1.

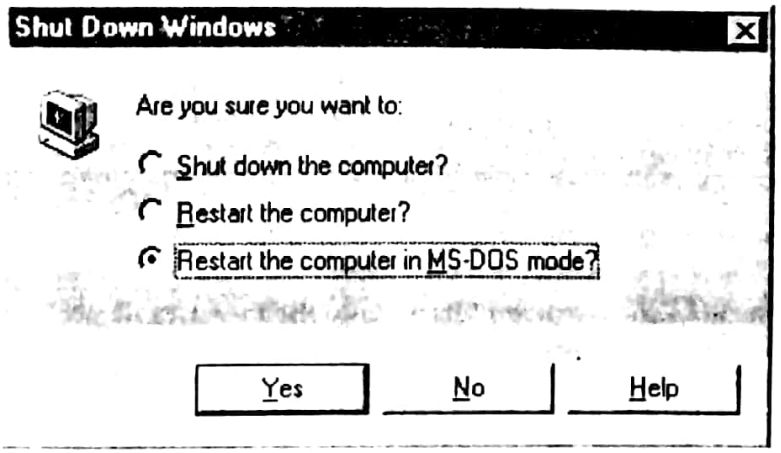


Figure 1

Durr-e-Danish Library

Scanned by CamScanner

```
#include<iostream.h>

void main()
{
cout<<"My first program in C++";
}
```

Header files contain various declarations describing library functions that you use to perform file access, mathematical computations, data conversion and many other standard functions. The line `#include<iostream.h>` instructs the compiler to include the declarations of the standard stream input and output facilities which are found in the header file iostream.h.

'`cout`' is defined in the header file '`iostream.h`'. It is used to display text and values on the screen. The operator << ("put to") writes value on its right-hand side to the left-hand side. In this case, the string '`My first program in C++`' is written onto the standard output stream `cout`.

When functions complete execution they may or may not return a value to the calling program. When a function returns a value, the data type of the value must be specified. If functions do not return any values to the calling program the return type is specified as `void`. The function `main()` has been defined as '`void`' since it does not return a value.

1.3 Saving the Program

To save the program, follow these steps:

1. **Press the key combinations, <Alt+F> to invoke the menu, 'File'.**
   This displays some options.

2. **Press the down arrow key to select the option, 'Save as...'.**
   Refer Figure 5.

Durr-e-Danish Library
Scanned by CamScanner

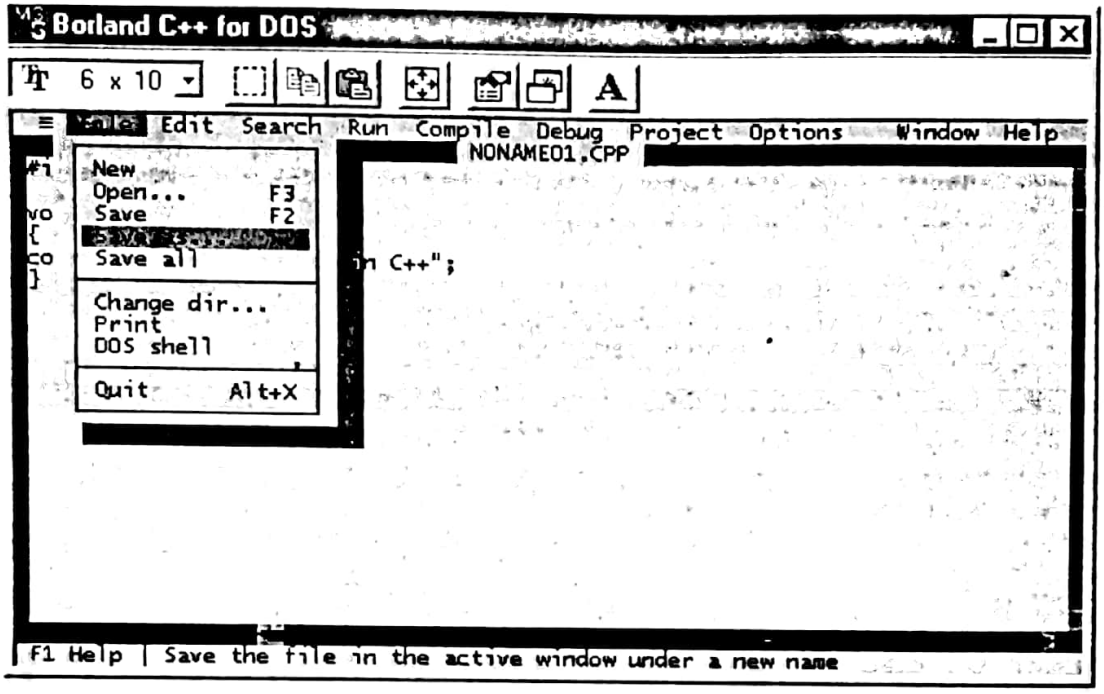


Figure 5

3. **Press <Enter>.**
   This displays the dialog box, 'Save File As'.

4. **Type FIRST as the filename in the text box, 'Save File As'.**
   Refer Figure 6.

Note: You should save the file in the working directory assigned to you.

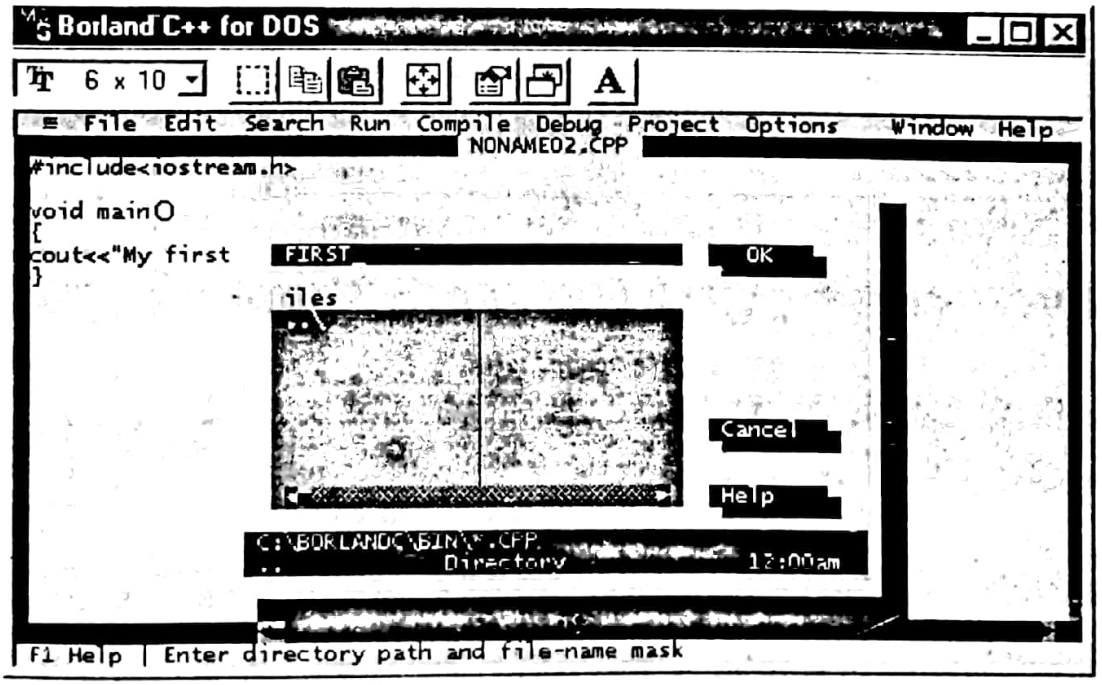


Figure 6

Durr-e-Danish Library
Scanned by CamScanner

5. **Press <Enter>.**
   This saves the program and the filename is displayed as FIRST.CPP in the title bar.
   Refer Figure 7.

Note: In your case the filename will be displayed along with the path of the folder where you are saving your file, e.g. C:\MYDIRECTORY\FIRST.CPP

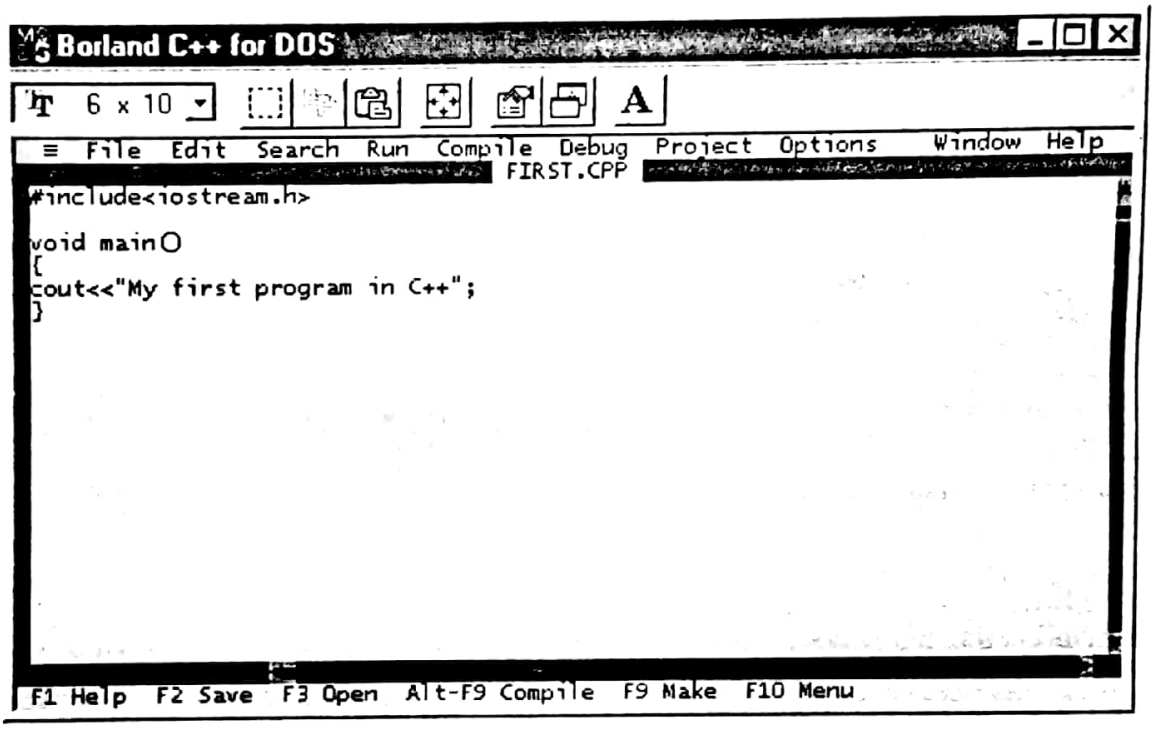


Figure 7

1.4 Building the Program

The program you type into the Edit window constitutes the source file. The build process that transforms the source file into an executable program requires two stages: compiling and linking. First the source file is compiled into an object file, and then linked to the necessary library routines.

To build the file, follow these steps:

1. **Press the key combination, <Alt+C> to invoke the 'Compile' menu.**
   This displays some options.

2. **Press the down arrow key to select the option 'Build all'.**
   Refer Figure 8. When you use the 'Build all' option the compiling and linking are done one after the other.

Instead of using the 'Build all' option you can,

- compile the program by selecting the 'Compile' option under the 'Compile' menu item (or press the <Alt+F9> key combination)
- link the program by selecting the 'Link' option under the 'Compile' menu item

Durr-e-Danish Library
Scanned by CamScanner

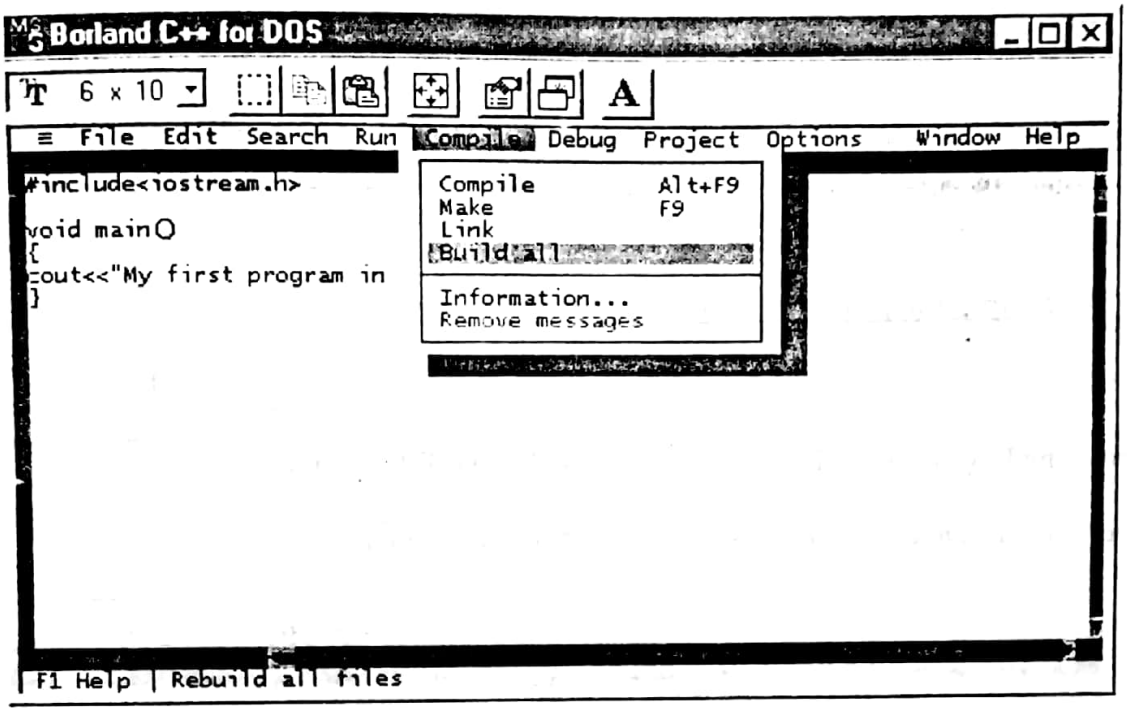


Figure 8

3. **Press <Enter>.**
   This builds the program. On success, the 'Linking' message box displays '0 Errors and 0 Warnings'. Refer Figure 9.

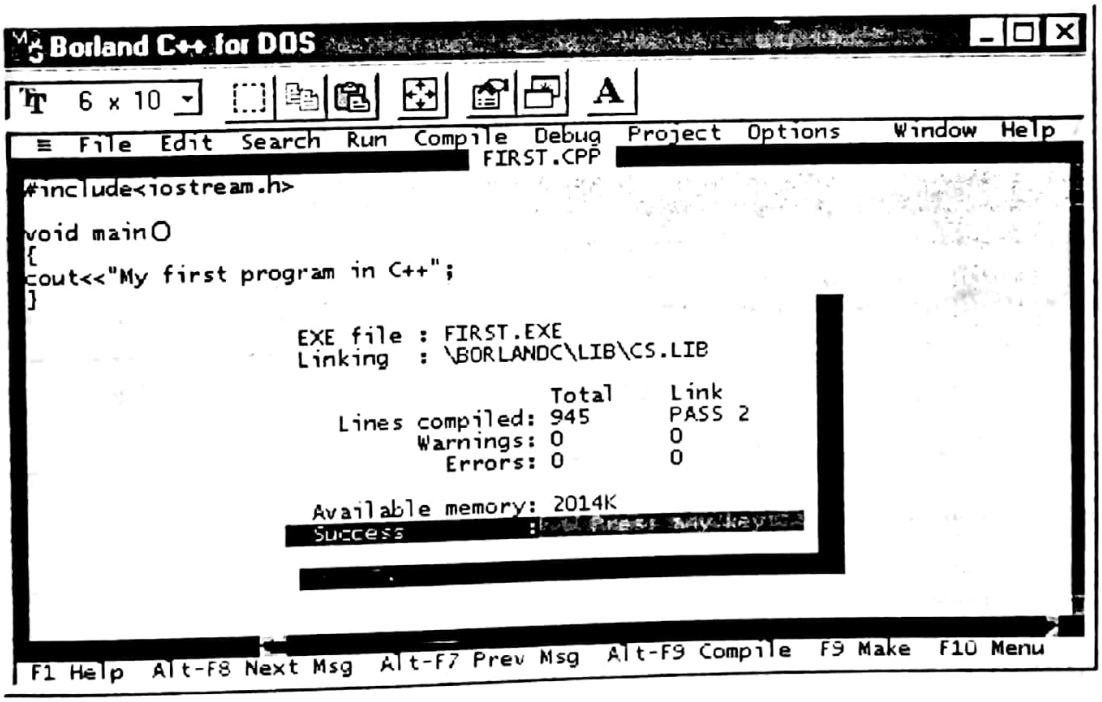


Figure 9

4. **Press the Esc key to close the message box.**

Durr-e-Danish Library
Scanned by CamScanner

1.5 Executing the Program

There are two ways to execute the program:

1. From the MS-DOS prompt
2. Using the Run menu options.

1.5.1 Executing the program from MS-DOS Prompt

To execute the program from the MS-DOS prompt and see the output, follow these steps:

1. **Press the key combination, <Alt+F> to invoke the 'File' menu.**

2. **Press the down arrow key to select the option, 'Dos shell'.**
   Refer Figure 10.

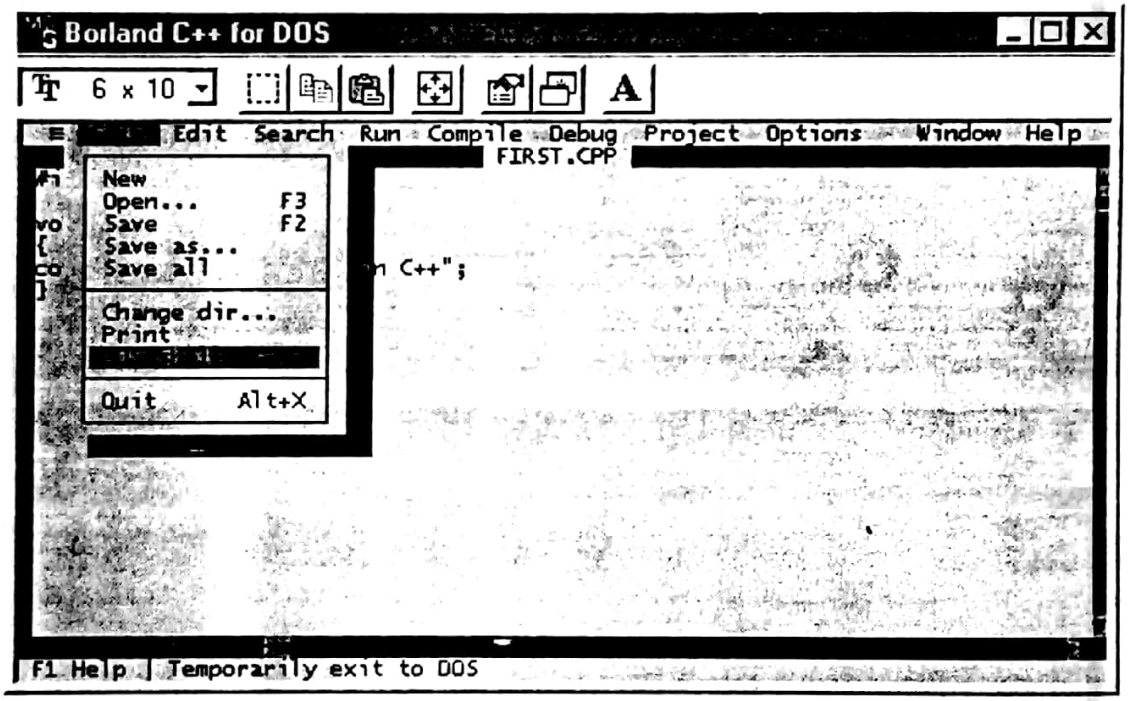


Figure 10

3. **Press <Enter>.**
   This takes you to the DOS prompt

```
C:\BORLANDC\BIN>
```

4. **Type FIRST.**
   This is the name of the file to be executed.

```
C:\BORLANDC\BIN>FIRST
```

5. **Press <Enter>.**
   This displays the output as follows:

Durr-e-Danish Library
Scanned by CamScanner

```
C:\BORLANDC\BIN>FIRST
My first program in C++
C:\BORLANDC\BIN>
```

6. **Type exit at the prompt.**

7. **Press <Enter>.**
   This takes you back to the Borland C++ editor.

- Another method to get to the MS-DOS prompt to run the program is to use the 'File' menu and 'Quit' option. This takes you to the MS-DOS prompt. You can run the program as described above. You can invoke the Borland C++ editor again after the program is executed.

1.5.2 Executing the program from the 'Run' menu

To execute the program from the 'Run' menu, follow these steps:

1. **Press the key combination, <Alt+R> to select the menu option, 'Run'.**

2. **Press the down arrow to select the option 'Run'.**
   Refer Figure 11. You will see the screen flicker briefly.

- You can also execute the program by pressing the <Ctrl+F9> key combination.

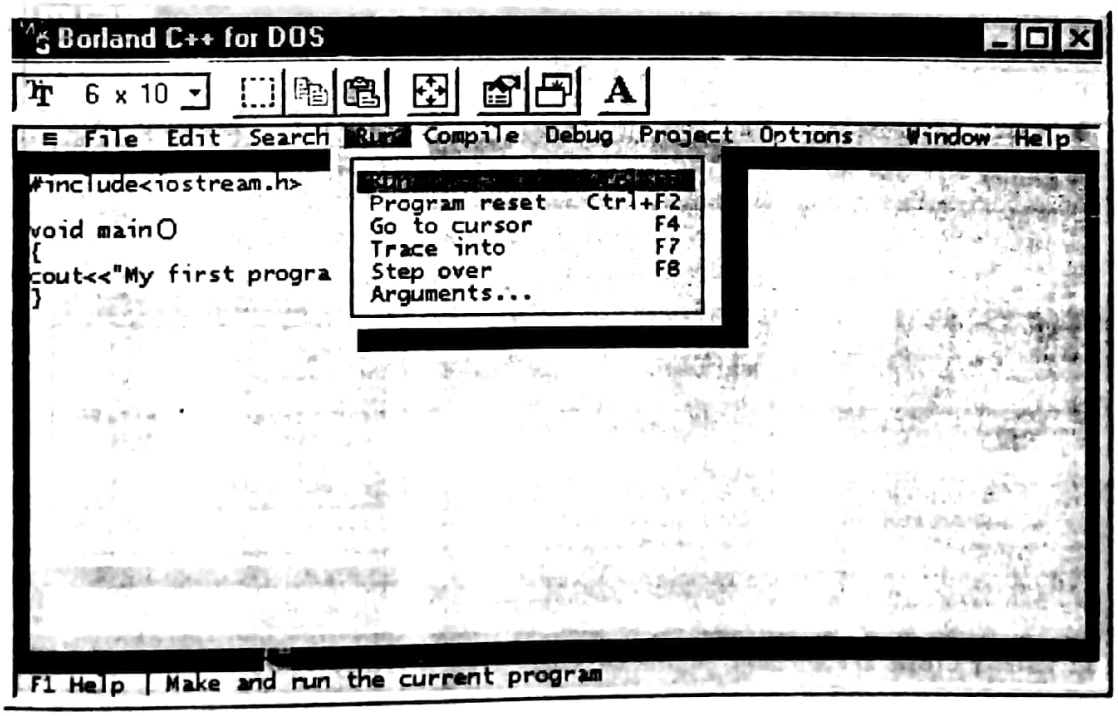


Figure 11

Durr-e-Danish Library
Scanned by CamScanner

To see the result:

3. Press the <Alt+F5> key combination.

Or

Select the option, 'User Screen' from the 'Window' menu and Press <Enter>.

The editor will vanish and you will see the normal MS-DOS screen, with the following display:

```
C:\BORLANDC\BIN>bc
My first program in C++
```

4. Press any key to return to the Borland C++ editor.

1.6 Closing the Window

To close the active window, follow these steps:

1. **Press the key combination, <Alt+W> to select the menu option, 'Window'.**
   This displays some options.

2. **Press the down arrow key to select the option, 'Close'.**
   Refer Figure 12.

You can also close the window that is currently open by using the <Alt+F3> key combination.

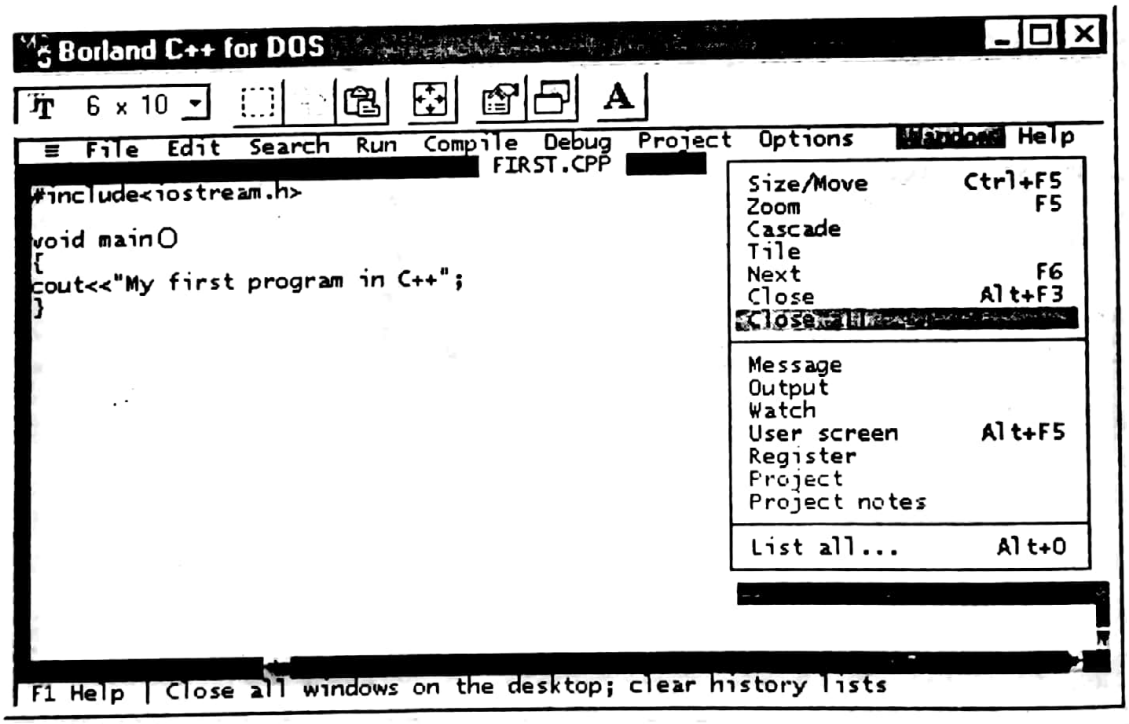


Figure 12

3. Press <Enter>.
This closes the active window. Refer Figure 13.

Durr-e-Danish Library

Scanned by CamScanner

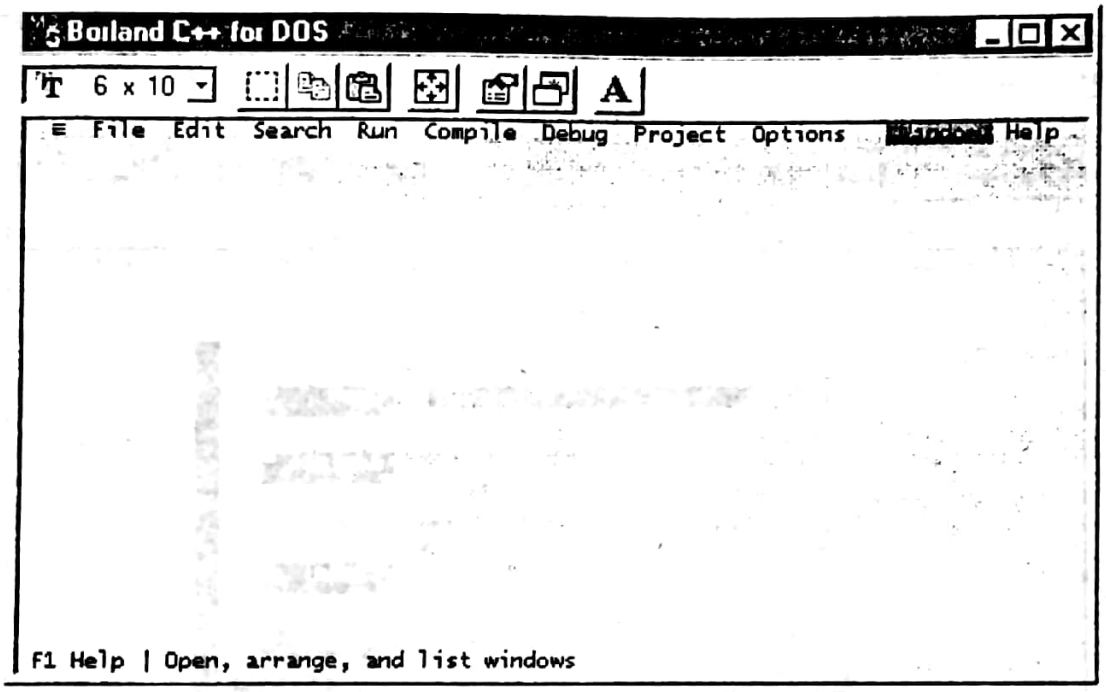


Figure 13

1.7 Opening an Existing File

To open an existing file, follow these steps:

1. **Press the <Alt+F> key combination to invoke the 'File' menu.**
2. **Press the down arrow key to select the option, 'Open...'.**
   Refer Figure 14.

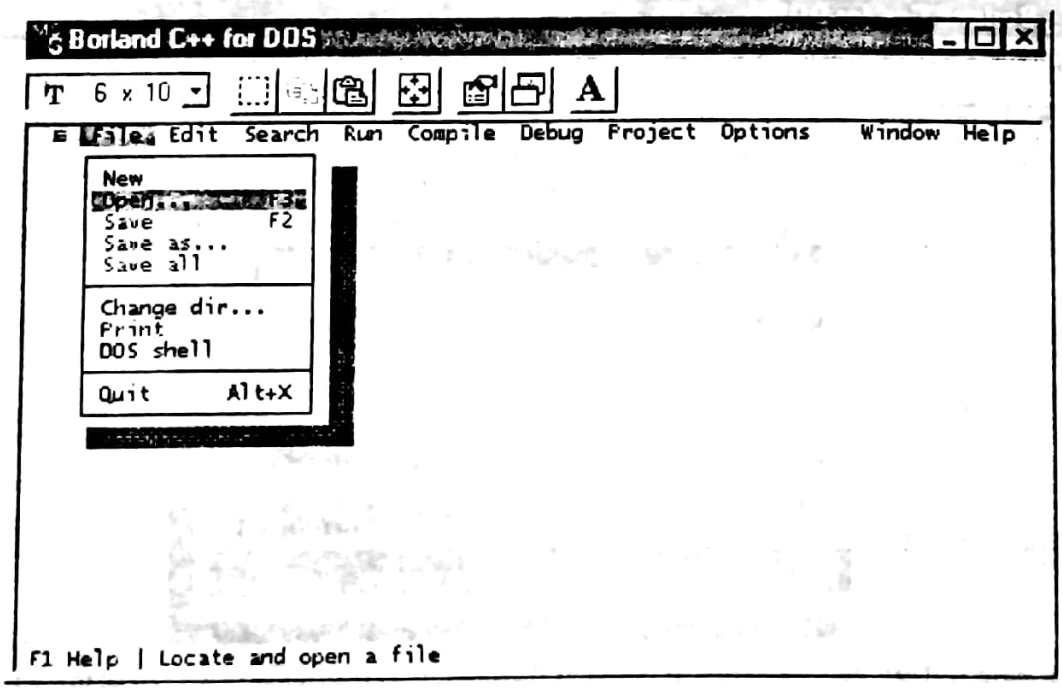


Figure 14

Durr-e-Danish Library
Scanned by CamScanner

3. Press <Enter>.
This displays the dialog box, 'Open a File'. Refer Figure 15.

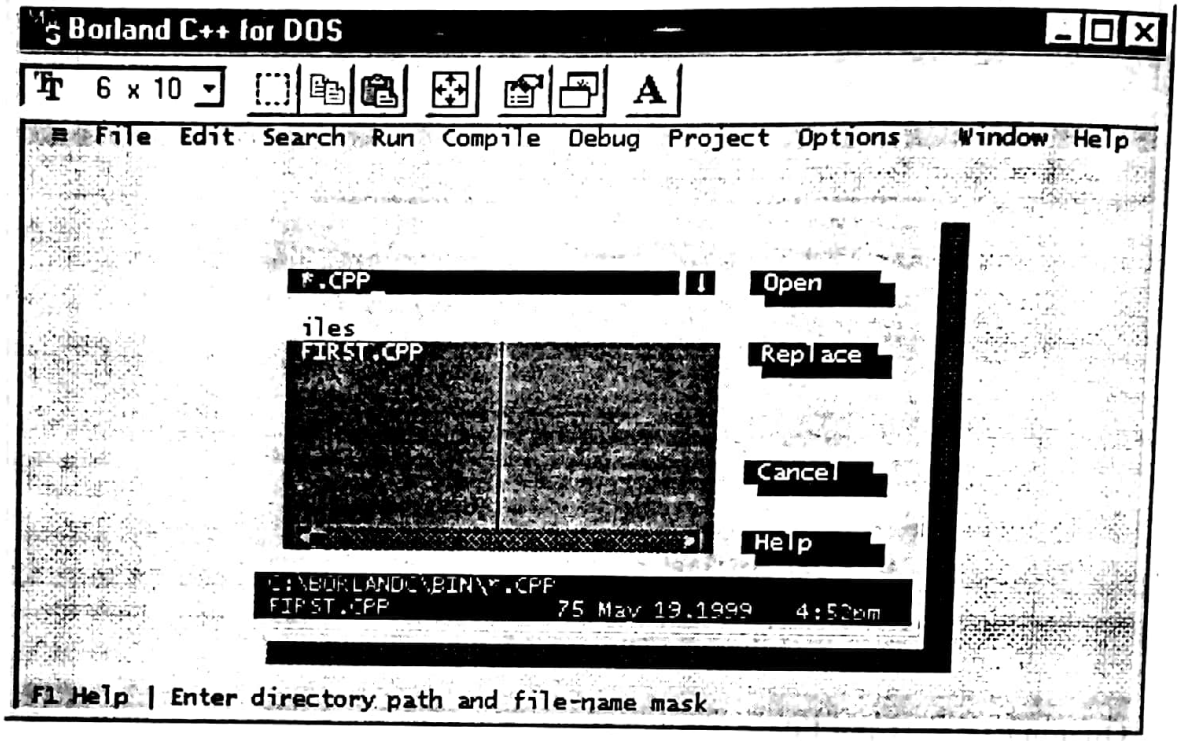


Figure 15

4. Press <Tab>.
The cursor moves to the box, 'Files'. The file FIRST.CPP is highlighted. Refer Figure 16.

You will have to go to your working folder to open the file saved as FIRST.CPP.

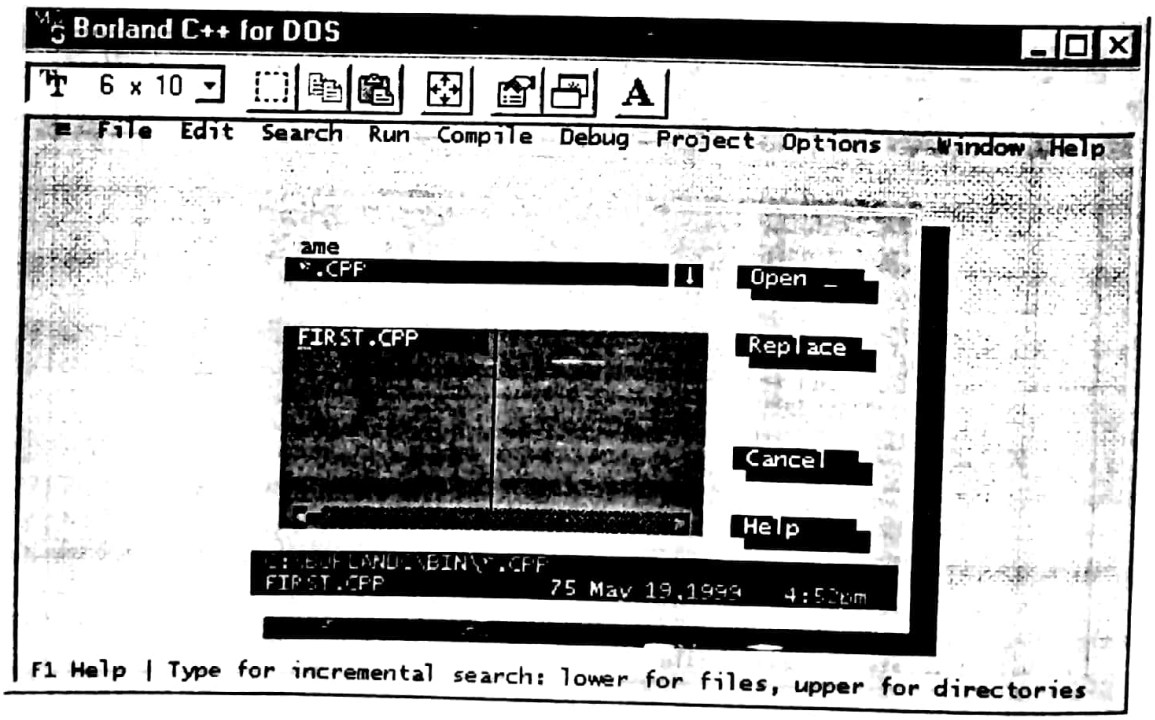


Figure 16

Durr-e-Danish Library
Scanned by CamScanner

5. **Press <Tab> to select the button, 'Open'.**

6. **Press <Enter>.**
   This opens the file and displays the code in the 'Edit window'.

7. **Close the file FIRST.CPP.**

1.8 Classes, Objects and Member Functions

In this section you will write programs to understand the following concepts:

- Classes
- Objects
- Member Functions and accessing them

1.8.1 Classes and Objects

The following program will help you understand how classes and objects are defined. In this program we will first write a class containing an integer variable and then define an object of the class in the main() program. The data variable will be assigned a value, which will be displayed on to the standard output.

1. **Create a new file.**

2. **Type the following code in the 'Edit' window.**

```
#include<iostream.h>

class myclass
{
 public:
      int data1;
};

void main()
{
      myclass object1;
      object1.data1 = 25;
      cout<< "\n The value of data1 is "<< object1.data1;
}
```

3. **Save the file as FILE1.CPP.**

4. **Build the file to get FILE1.EXE.**

5. **Execute the program, FILE1.**

Durr-e-Danish Library
Scanned by CamScanner

The above program displays a value,

OUTPUT:

```
The value of data1 is 25
```

The above program has a class named `myclass`. A variable `data1` of the type integer is defined in this class as a public variable.

In `main()`, an object named `object1` of the type `myclass` is created. A value 25 is assigned to the variable `data1` of the object, `object1`. The 'cout' prints the assigned value. '\n' is used to display the output on the next line.

- Since the variable `data1` has been defined as a public variable it is possible to assign a value directly in the `main()` program. If it were a private data member, you would be able to access it only in the member function.

6. Close the file, FILE1.CPP.

1.8.2 Accepting, Displaying and manipulating public data members

This is another program that explains the concepts of class and object. It accepts your name and age as input from the standard input and displays it on the standard output. It also checks the person's age and displays a message as specified below:

Age	Message
<100	You are pretty young
=100	You are old
>100	You are really old

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>

class person
{
 public:
char name[16];
int age;
};                  // end of class person

void main()
{
      person myself;

      cout << "\n Enter your name:  ";
      cin >>myself.name;
```

Durr-e-Danish Library

Scanned by CamScanner

```
      cout << "\n Your name is " << myself.name;
      cout << "\n Enter your age:  ";
      cin >> myself.age;
      cout<< "\n You are " << myself.age << " years old";

      if(myself.age < 100) {cout<<"\n You are pretty young!"; }
      if(myself.age == 100){cout<<"\n You are old"; }
      if(myself.age > 100) {cout<<"\n You are really old"; }
}
```

3. **Save the file as FILE2.CPP.**

4. **Build the file to get FILE2.EXE.**

5. **Execute the program, FILE2.**
   This displays the following output and prompts you to enter your name.

```
Enter your name:
```

6. **Type BILL at the prompt and press <Enter>.**

After you type the name and press the <Enter> key it will display the name that was input and again prompt you to enter your age as shown below.

```
Your name is BILL
Enter your age:
```

7. **Type 23 at the prompt as shown below and press <Enter>.**

```
Enter your age: 23
```

After you type the age and press the <Enter> key it will display the age that was input and also a message depending on the age.

```
You are 23 years old
You are pretty young!
```

The above program has a class named `person`. The variable `age`, of the type '`int`', and `name`, of the data type '`char`', are defined in this class. In `main()`, an object, `myself`, is created from `person`. The program reads your name and age and displays the output.

'`cin`' accepts input from the user. It is defined in the header file, '`iostream.h`'.

'\n' displays the output on the next line. In C++, comments are prefaced by //. All characters following // are treated as part of the comment.

8. **Close the file, FILE2.CPP.**

Durr-e-Danish Library
Scanned by CamScanner

1.8.3 Accessing Member Functions

In this program we will write a class which has a private data member and public member functions. We will see how the member functions are used to access the private data member of the objects created and assign values.

1. **Create a new file.**

2. **Type the following code in the 'Edit' window.**

```
#include <iostream.h>

class Hotel_rooms
{
   private:
      int room_no;

      public:

      void set(int invalue)
      {
      room_no = invalue;
      }

      int getvalue()
      {
      return room_no;
      }

};

void main()
{
      Hotel_rooms room1, room2, room3;
      int suite_no;

   room1.set(12);
   room2.set(17);
   room3.set(13);
   suite_no = 123;

   cout << "The number of room1 is " << room1.get_value()<< "\n";
   cout << "The number of room2 is " << room2.get_value()<< "\n";
   cout << "The number of room3 is " << room3.get_value()<< "\n";
   cout << "The number of suite_no is " << suite_no << "\n";
}
```

3. **Save the file as FILE4.CPP.**

4. **Build the file to get FILE4.EXE.**

Durr-e-Danish Library

Scanned by CamScanner

5. Execute the program, FILE4.

OUTPUT:

```
The number of room1 is 12
The number of room2 is 17
The number of room3 is 13
The number of suite_no is 123
```

The above program has a class named `Hotel_rooms`. The variable `room_no` is of the type `int` and is a private data member. The function `set()` has one parameter `invalue` of type `int` that is used to set the value of the variable `room_no`. The function `getvalue()` returns a variable of type `int` that is used to display the value of `room_no`. This function has no parameters. Both the functions are defined in the class in the public section.

In `main()`, three objects, `room1`, `room2`, `room3`, are created of the class `Hotel_rooms`. There also is one integer variable `suite_no` defined in `main()`. Observe the difference in the way the values are assigned to the data member of the objects created and to the variable `suite_no`. The function `getvalue` is used to display the value of the data member.

6. Close the file, FILE4.CPP.

1.9 Exiting Borland C++

To exit Borland C+ follow these steps:

1. **Press the key combination, <Alt +F> to invoke the 'File' menu.**
2. **Press the down arrow key to select the option 'Quit'**
   Refer Figure 17.

Durr-e-Danish Library
Scanned by CamScanner

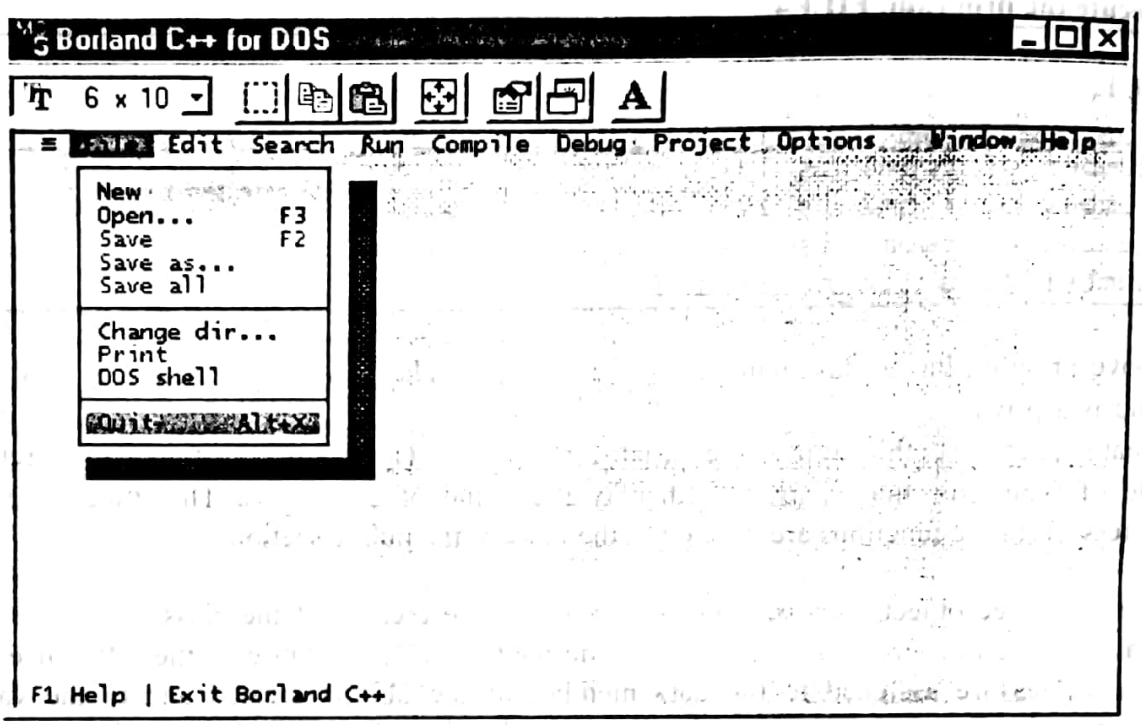


Figure 17

3. **Press <Enter>.**
   This brings you back to the DOS prompt.

Durr-e-Danish Library
Scanned by CamScanner

The Next 1 Hour:

Try out the following exercises:

1. Write a program to calculate the area of a rectangle given its height and width. If the height and width are equal you should state that it is a square.

2. Write a program to read the name and age of five students, and print the name, age and the average student age.

3. Write a program to create a class called Number with n1 and n2 as two data member variables of integer type. It should contain member functions to read the numbers, add, subtract, multiply and divide the two numbers, and display the results.

4. Write a program to prepare the mark sheet of a student. The following items may be read in from the keyboard:

- Name of the student
- Roll number
- Subject_id
- Subject marks

Assume that the examinations are in at least five subjects. Your program should display the marks. It should also contain member functions to calculate and display the total and average marks of the five subjects. You can use the following class structure:

```
class Student{
 private:
   char name[25];
   int rollno;
   int sub_id[5];
   int sub_marks[5];
 public:
   getinfo();
   total();
   display();
};
```

Hint: You can use a For loop or a Do---While loop to accept values for the five subjects and marks.

Durr-e-Danish Library

Scanned by CamScanner

This page has been intentionally left blank

Durr-e-Danish Library
Scanned by CamScanner

Session 2

Session Objectives :

At the end of this session, the student will be able to -

- *Write programs that reinforce the concepts of*
  - *Classes and Objects*
  - *Public and Private Member Data and Functions*
  - *Arrays of Class Objects*

The steps given in the session are detailed, comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

The First 1 Hour:

2.1 Classes, Objects and Member Functions

In this section you will write some more programs which will help reinforce the concepts of:

- Classes and Objects
- Public and Private member data and functions

2.1.1 Public Data Members in Classes

The program below is used to display the date. The date is assigned a value in the `main()` program. This program and the next will demonstrate the accessibility of the data members of a class when they are declared as public or private.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>

class date
{
 public:
      int day;
      int month;
      int year;
};

void main()
{
      date today;
      today.day = 10;
      today.month = 8;
      today.year = 1999;
```


Durr-e-Danish Library

Scanned by CamScanner

```
        cout << "\n The date is "<< today.day << "/";
        cout << today.month << "/" << today.year << endl;
}
```

3. Save the file as DATE1.CPP.

4. Build the file to get DATE1.EXE.

5. Execute the program, DATE1.

OUTPUT:

```
The date is 10/8/1999
```

In the above program a class named `date` is declared. It contains three public variables of integer type.

Since the data members are declared public, they can be accessed in the `main()` program. Hence, no member function is used in this program.

☞ The 'endl' is an output manipulator to generate a carriage return or line feed character. It is similar to the character '\n' that is used to generate line feed.

6. Close the file, DATE1.CPP.

2.1.2 Accessibility of Private Data Members in a Class

If the keyword public is not used to define the members of a class, the C++ compiler assumes, by default, that all its members are private. Such private data members are not accessible outside the class. In this program we will try to display the date as in the previous program, but with the use of private data members.

1. Modify the file DATE1.CPP from the previous program as follows.

```
#include<iostream.h>

class date
{                          // by default, members are private
      int day;
      int month;
      int year;
};
void main()
{
      date today;

      today.day = 10;
      today.month = 8;
      today.year = 1999;
      cout << "\n The date is "<< today.day << "/";
```


Durr-e-Danish Library
Scanned by CamScanner

```
    cout << today.month << "/" << today.year << endl;
}
```

2. Save the file as DATE2.CPP.

3. Build the file.
You will receive the following error message,

ERROR MESSAGE:

```
date:: day is not accessible
date:: month is not accessible
date:: year is not accessible
```

The above program is similar to the previous one except that the class data members are private data members. Though the keyword private has not been included by default the variables are treated as private. You can see from the error message that the accessibility of the data variables changes when they are treated as private members of the class. To access the private data members you will need to use member functions.

4. Close the file, DATE2.CPP.

2.1.3 Using Member Functions to Access Private Data Members

The program listed below is also used to display the date. The value for the date that is input by the user is displayed. We will see how this is done using member functions to receive input and display the results.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include <iostream.h>

class date
{
 private:
      int day;
      int month;
      int year;
public:
      void getdate(void)
      {
            cout << "\n Enter the day (dd): ";
            cin >> day;
            cout << "\n Enter the month (mm): ";
            cin >> month;
            cout << "\n Enter the year (yy): ";
            cin >> year;
```


Durr-e-Danish Library
Scanned by CamScanner

```
    }           // end of function getdate

    void display(void)
    {
          cout << "\n The date is "<< day << "/";
          cout << month << "/" << year << endl;
    }           // end of function display

};

void main()
{
     date today;

today.getdata();
today.display();
}
```

3. Save the file as DATE3.CPP.

4. Build the file to get DATE3.EXE.

5. Execute the program, DATE3.

It will prompt you to enter the day, month and year one by one and then display the date as shown below.

OUTPUT:

```
Enter the day (dd): 10
Enter the month (mm): 8
Enter the year (yy): 99
The date is 10/8/99
```

The above program has a class named `date`. The variables `day, month and year`, are of the type integer and are private data members. The functions `getdate()`, and `display()` are also defined in the class, in the public section.

In `main()`, an object, `today`, of the type `date` is defined. The program reads the date using the function `getdate()` and the function `display()` displays the date.

Note that the output will vary depending on the date you have entered.

In `main()`, the object `today` and the member functions are connected using the dot operator or the class member operator. Compare this program with the previous one to observe how the private data members are accessed correctly using the member functions.

6. Close the file, DATE3.CPP.

Durr-e-Danish Library

Scanned by CamScanner

2.1.4 Passing Objects as arguments to Member Functions

This is a program that uses member functions to read the data variables of a class and display the contents of the class objects. Here we will use a member function that accepts objects as parameters.

In this program we will create a class `Time` with hours, minutes and seconds as data members. We will create two objects of type `Time`. The two objects will be added using a member function. In this member function we will perform a check to see if the added minutes and seconds are greater than 59. If the seconds are greater than 59, the minutes will be incremented by one. Similarly the hours will be incremented by one if the minutes are greater than 59.

1. **Create a new file.**

2. **Type the following code in the 'Edit' window.**

```
#include <iostream.h>

class Time
{
   private:
    int hours, minutes, seconds;
   public:
    void getinfo()
    {
    cin >> hours>>minutes>>seconds;
   }

   void display()
   {
     cout << hours << ":" << minutes << ":"<< seconds << endl;
   }

   void addit(Time aa, Time bb)
   {
      hours = aa.hours + bb.hours;
      minutes = aa.minutes + bb.minutes;
      seconds = aa.seconds + bb.seconds;
     // checking that seconds is < 60
     if ( seconds >= 60 )
     { seconds -= 60;
       minutes++;
     }
     // checking that minutes < 60
     if ( minutes >= 60)
     {
       minutes -= 60;
       hours++;
     }
   }
};
```


Durr-e-Danish Library
Scanned by CamScanner

```
void main()
{
  Time one, two, three;
     cout<< "\nEnter the first time (hours minutes seconds): ";
     one.getinfo();
     cout<< "Enter the second time (hours minutes seconds): ";
     two.getinfo();
     three.addit(one,two);
     cout << "The result is ";
     three.display();
}
```

3. **Save the file as TIME.CPP.**
4. **Build the file to get TIME.EXE.**
5. **Execute the program, TIME.**

It will prompt you to enter the time.

Note: While entering the time separate the values for the hours, minutes, and seconds using the spacebar.

OUTPUT:

```
Enter the first time (hours minutes seconds): 14 45 15
Enter the second time (hours minutes seconds): 3 19 14
The result is 18:4:29
```

The above program has a class named `Time`. The variables `hours`, `minutes` and `seconds` are of the type integer and are private data members. This program again clearly brings out the concept of how objects access member functions of the class. In `main()`, three objects of the class `Time` are defined, namely, `one`, `two` and `three`.

The function `addit()` is used to add the two `Time` objects passed to it as arguments. Here, `one` and `two` objects are passed as arguments as shown below.

```
third.addit(one,two);
```

actual arguments

Objects `one` and `two` are accessed through `aa` and `bb` objects respectively in the definition of the member function.

```
void addit(Time aa, Time bb);
```

formal arguments

The member function `addit()` can access the private data members of these objects passed as arguments by.

Durr-e-Danish Library
Scanned by CamScanner

```
aa.hour;
```

The member function `addit()` in the main program is being invoked by the object `third` and is called the invoking object.

```
third.addit(one, two);
```

invoking object

In the function definition of `addit()`, observe the statement

```
hours = aa.hours + bb.hours;
```

Data member of object `third`

Here `hours` refer to the data member of the invoking object `third`. Since the added hours, members and seconds are being stored in the invoking object it is not necessary to return the resultant object.

6. Close the file, TIME.CPP.

2.2 Arrays of Objects

Arrays are user defined data types whose members are of the same type. For practical applications such as designing a large database, arrays are very useful. Arrays of class objects are used in a manner similar to other basic data types.

We will write a program that will define a class Student with the roll number and marks as the data members. The average marks obtained by the students are calculated and displayed along with the student details.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include <iostream.h>

class Student
{
   private:
      int rollno;
      int marks;
   public:
      void getinfo()
      {
        cout<<"Enter the Roll number:";
        cin>>rollno;
        cout<<"Enter the marks:";
        cin>>marks;
```


Durr-e-Danish Library

Scanned by CamScanner

```
    }
    void display()
    {
       cout << rollno <<'\t'<< marks<<endl;
    }
    int getmarks()
    {
     return marks;
    }
};        //end of class Student

void main()
{
    Student stulist[100];
    float total, average;
    int no,i;

    total = 0.0;

    cout << "Enter the number of students:";
    cin>>no;

    for (i=0;i<no;i++)
    {
     stulist[i].getinfo();
     total = total+stulist[i].getmarks();   //adding the marks
    }

   cout <<"Rollno  Marks"<<endl;
     for (i=0;i<no;i++)
    {
     stulist[i].display();
    }
    average = total/no;                     //find the average
    cout<< "Average marks = "<<average;

}              //end of main
```

3. **Save the file as STUD1.CPP.**

4. **Build the file to get STUD1.EXE.**

5. **Execute the program, STUD1.**

It will prompt you to enter the number of students for whom details will be supplied. You will also be prompted to enter the roll number and marks of the students as shown below.

Durr-e-Danish Library

Scanned by CamScanner

OUTPUT:

```
Enter the number of students: 3
Enter the Roll number: 1
Enter the marks: 23
Enter the Roll number: 2
Enter the marks: 25
Enter the Roll number: 3
Enter the marks: 20
Rollno Marks
1    23
2    25
3    20
Average marks = 22.666667
```

In the above program a class `Student` is defined with `rollno` and `marks` as private data members of type integer.

In `main()`, an array of hundred student objects are defined using the statement `stulist[100]`. The user is asked to enter the number of students for whom details will be given. The user is also prompted to enter these details one by one.

Observe the following statements:

```
for (i=0;i<no;i++)
{
   stulist[i].getinfo();
   total = total+stulist[i].getmarks();
}
```

Here, `stulist[i].getinfo();` calls the member function `getinfo()` to accept the student details. The subscript `i` changes from 1 to the total number of students. Hence, one by one all student details are accepted.

`getmarks()` is another member function that returns the marks of the invoking object which is added to the variable `total`. Thus, at the end of the for loop the variable `total` will have the sum of the marks obtained by the students. Finally the average is calculated and displayed along with the student details.

In the member function `display()`, '\t' is used to insert a tab between the variables.

Thus, this program clearly shows how to create an array of objects and how the array objects access the member functions.

Durr-e-Danish Library

Scanned by CamScanner

The Next 1 Hour:

Try out the following exercises:

1. Rewrite the program in FILE2.CPP of Session 1 using member functions to get information regarding the person's name and age, and display the information.

2. Write a program to read in a person's date of birth in the format "date month year" and display it in the format shown. For example if the person's date of birth is 23 10 1959 the output should read "23 October, 1959". You can use a class Date with data members, date, month and year, and member functions to accept values and display the result in the required format.
   Hint: Use switch and case statements to correlate the numbers with the twelve months of the year.

3. Write a program to read an integer number and find out the sum of all the digits until it is reduced to a single digit. For example,

   If the number is 1256
   Sum = 1+ 2 + 5+6 = 14
   Sum = 1+4 = 5

   You can use the following class structure with an integer data member and member functions to accept the number from the user, perform the addition, and display the result.

```
Class Number{
   int num;
   public:
         getinfo();
         add_digits();
         display();
);
```

The result can be displayed in the following format:

```
The sum of the digits of 1256 is 5
```

❖ ❖ ❖ ❖ ❖

Durr-e-Danish Library

Scanned by CamScanner

Session 3

Session Objectives :

At the end of this session, the student will be able to -

- *Write programs that utilise*
  - *Constructors*
  - *Destructors*
  - *Function overloading*
  - *Friend functions*

The steps given in the session are detailed, comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

The First 1 Hour:

3.1 Constructors and Destructors

In this section you will write some programs which will help reinforce the concepts of:

- Constructors
- Destructors

Constructors are used to automatically initialise an object when it is created, without the need to make a separate call to a member function. Following the same principle, a destructor is a function that is automatically called when an object is destroyed.

3.1.1 A simple Constructor and destructor

In the program below we will see how constructors and destructors are implemented. The program will be used to generate the area of a rectangle, given the length and width. The program will also make use of the scope resolution operator to define member functions outside the class specifier.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>
#include<conio.h>

class rectangle
{                              // A simple class
   private:
      int height;
      int width;
  public:
   rectangle();                // a constructor
```


Durr-e-Danish Library

Scanned by CamScanner

```
    int area();
    void initialise(int, int);
    ~rectangle();              // and a destructor
};

rectangle::rectangle()     // constructor
{
    cout << "Rectangle constructor in action\n";
    height = 6;
    width = 6;
}

int rectangle::area()      //Area of a rectangle
{
    return height * width;
}

void rectangle::initialise(int init_height, int init_width)
{
    cout << "Initialising rectangle with given values\n";
    height = init_height;
    width = init_width;
}

rectangle::~rectangle()    // destructor
{
    cout << "Rectangle destructor in action\n";
}

void main()
{
    clrscr();                  //to clear screen
    rectangle box, square;
    int can_ht,can_width, can_area;

    cout<<"The area of the box is "<<box.area() << "\n";
    cout<<"The area of the square is "<<square.area() << "\n";

    box.initialise(12, 10);
    square.initialise(8, 8);
    cout<<"The area of the box is "<<box.area() << "\n";
    cout<<"The area of the square is "<<square.area() << "\n";

    can_ht = 50;
    can_width = 6;
    can_area = can_ht * can_width;
    cout << "The area of the can is " <<can_area<< "\n";

}
```

3. Save the file as CON1.CPP.

Durr-e-Danish Library

Scanned by CamScanner

4. **Build the file to get CON1.EXE.**

5. **Execute the program, CON1.**

OUTPUT:

```
Rectangle constructor in action
Rectangle constructor in action
The area of the box is 36
The area of the square is 36
Initialising rectangle with given values
Initialising rectangle with given values
The area of the box is 120
The area of the square is 64
The area of the can is 300
Rectangle destructor in action
Rectangle destructor in action
```

In the above program a class named `rectangle` is declared. It contains two private variables of integer type. The public member functions include a constructor and a destructor. The constructor is given the same name as the class in which it operates. The destructor also has the same name except that it is preceded by a tilde (~). Each time an object is created in `main()` the constructor is invoked. All the functions declared in the class have been defined outside of the class.

- Note the use of the scope resolution operator :: to *define* the functions outside the class specifier, after they have been *declared* within the class. By giving the name of the class to the left of the scope resolution operator we tell the compiler which class the function belongs to.

In this program we have included a standard header file `conio.h`. In `main()` before any declarations are made we have called the function `clrscr()`. This function is defined in `conio.h`. This function clears the standard output screen of all text and places the cursor at the upper left-hand corner of the screen ready to receive the output from the current program. Only the output from the current program will then be displayed on the screen. This function is useful to keep the screen free of any text that it contained previous to the execution of the current program.

In `main()` the statement

```
rectangle box, square;
```

creates two objects of type `rectangle`. We have added a statement in the constructor that is displayed every time an object is created. In the program output you can see the statement

```
Rectangle constructor in action
```

It is displayed twice denoting that two objects have been created. The constructor in this program initialises the `rectangle` objects with the default value of 6 for the length and width. When

Durr-e-Danish Library

Scanned by CamScanner

the function area() is called the first time to calculate the area of the box or square it is these default values that are used.

In addition there is also a member function initialise(), which is used to give values to length and width. This function is called when we use the statements

```
box.initialise(12,10);
square.initialise(8,8);
```

The default values given to box and square by the constructor are replaced when the function initialise() is called. In the program output, the statement

```
Initialising rectangle with given values
```

is displayed to show that the function initialise() is called.

The area of a can is also calculated in main(), using local variables can_ht, can-width and can_area. This calculation has been given in the program to make you note the difference in the way the class data members and objects are used as opposed to local variables.

When the objects box and square are destroyed at the end of main(), the destructor function is automatically called. We can see that the destructor is invoked, when we see the following statement in the program output:

```
Rectangle destructor in action
```

The statement is displayed twice at the end of the output indicating that the two objects that were created are destroyed.

6. Close the file, CON1.CPP.

3.1.2 More on Constructors and Destructors

In the program below we will use the file TIME.CPP that was created in Session 2. In this program we will use constructors to initialise the values of the data members hours, minutes and seconds of the class Time. You will see how constructors can be used with different arguments or no arguments to create objects. The program will take two Time objects, add them and display the result. A destructor will also be used to destroy the objects.

1. Modify the file TIME.CPP that was created in Session2 as follows

```
#include<iostream.h>
#include<conio.h>

class Time
{
   private:
        int hours, minutes, seconds;
   public:
    Time()                    //constructors
```


Durr-e-Danish Library
Scanned by CamScanner

```
    {
    hours=minutes=seconds=0;
    cout << "In default constructor \n";
    }
  Time(int h, int m,int s)
    {
    hours=h; minutes=m; seconds = s;
    cout << "Constructor with three arguments \n";
    }

  void display()
  {
    cout << hours << ':' << minutes << ':'<< seconds << endl;
  }
      operator +
  void addit(Time aa, Time bb)
  {
    hours = aa.hours + bb.hours;
    minutes = aa.minutes + bb.minutes;
    seconds = aa.seconds + bb.seconds;
    // checking that seconds is < 60
    if ( seconds >= 60 )
    { seconds -= 60;
      minutes++;
    }
    // checking that minutes < 60
    if ( minutes >= 60)
    {
      minutes -= 60;
      hours++;
    }
  }
  ~Time()                          //destructor
  {
    cout << "In destructor\n";
  }
};

void main()
{
  clrscr();
  Time result;
  Time one(1,49,45);
  Time two(3,40,30);

  result.addit(one,two);
  cout << "The result is ";
  result.display();
}
```

temp
temp.hour = hour + aa.hours.
temp.min = min + aa.
return temp;

2. Save the file as TIME2.CPP.

Durr-e-Danish Library
Scanned by CamScanner

3. **Build the file to get TIME2.EXE.**

4. **Execute the program, TIME2.**

OUTPUT:

```
In default constructor
Constructor with three arguments
Constructor with three arguments
In destructor
In destructor
The result is 5:30:15
In destructor
In destructor
In destructor
```

The above program uses two constructors to create objects. One of the constructors does not take any arguments. This is the default constructor and it takes no arguments. The second constructor takes three integer arguments. In `main()`, with the following statement,

```
Time result;
```

the object `result` is being created and thus it automatically invokes the default constructor `Time()`. In the output we see that this constructor is called by displaying the statement.

```
In default constructor
```

In this default constructor the value 0 is assigned to the data members hours, minutes, and seconds.

When we create objects using the following statements,

```
Time one(1,49,45);
Time two(3,40,30);
```

we are in fact calling the second constructor that takes three integer arguments. The values passed are assigned to the data members. The following statement is displayed twice indicating that two objects are created with this constructor:

```
Constructor with three arguments
```

Following this statement we see that the destructor has been called. This may seem mysterious till you understand how this has occurred even before the program has finished execution. So far we have seen that a destructor is invoked every time an object is destroyed. In this program the two arguments, `aa` and `bb`, that are taken by the function `addit()` are of the type `Time`. Note that a destructor is also invoked at the end of any function that contains an argument of the class or any object that is locally declared in the function. This is the reason that the statement "`In destructor`" appears twice in the output before the result is displayed.

Durr-e-Danish Library

Scanned by CamScanner

The functions for adding the Time objects and displaying the result are similar to those in the file TIME.CPP. At the end of main() all three of the objects are destroyed. Corresponding to this automatic action, the statement "In destructor" is displayed thrice at the end of the output. Destructors do not have return values and take no arguments since the assumption is that there is only one way to destroy an object.

In the above program, by using constructors that have different numbers of arguments we are, in fact, overloading the constructor function.

5. Close the file, TIME2.CPP.

3.1.3 Constructor and Destructor with dynamic allocation

In C++ the new operator often appears in constructors to allocate memory while initialising an object. This memory has to be deallocated. This is usually done using the delete operator in a destructor.

In the program below we will use a class that contains a data member which is a string. We will use a constructor to allocate memory for the string using the new operator. The destructor of the class will delete the memory space that is allocated to the string data member.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>
#include<string.h>
#include<conio.h>

class Command
{
 private:
    char* name;
 public:
    Command(char* s)                //constructor
    {
       int size = strlen(s);        //length of string argument
       name = new char[size+1];     //allocate memory
       strcpy(name,s);              //copy argument to name
    }
    void display();
    ~Command()                      //destructor
    { delete[] name;}
 };
void Command::display()
{
    cout<<name<<endl;
    cout<<"Length of the command string = "<<strlen(name)<<endl;
}
```


Durr-e-Danish Library

Scanned by CamScanner

```
void main()
{
    clrscr();
    Command first("Start of program");
    Command second("Stopping");

  first.display();
  second.display();
}
```

3. **Save the file as CONNEW.CPP.**

4. **Build the file to get CONNEW.EXE.**

5. **Execute the program, CONNEW.**

OUTPUT:

```
Start of program
Length of the command string = 16
Stopping
Length of the command string = 8
```

In the program the class `Command` has only one data member, a pointer to `char`, called `name`. This pointer will point to the string that will be created dynamically at run-time. Since we are making use of standard string functions the `string.h` header file has been included in the program.

The constructor takes a string as an argument. It creates a memory area for the string using the `new` operator to create the string. The function `strlen()` is a standard function that finds the length of a string and assigns it to the private data variable `size`. The length of the string is used while creating a new string as you can see in the statement,

```
name = new char[size+1];
```

The constructor copies the string into this area that is called `name`. It uses the standard `strcpy()` function to copy the string to `name`.

You will notice that `size` has been incremented by one while creating the new string. When it calculates the length of a string the `strlen()` function does not include the null ('\0') that terminates a string. However, while creating memory for the new string, space should be provided for the null character, which is added to any string. While displaying a string the << keeps printing characters until it encounters a '\0'.

The `display()` function outputs the string along with the length of the string using the standard `strlen()` function.

When we allocate memory using the new operator it is essential that we have a destructor which will deallocate that memory when it is no longer needed by the program. The destructor uses `delete[]` to free the string `name` that was created using the `new` operator.

Durr-e-Dahish Library

Scanned by CamScanner

6. **Close the file, CONNEW.CPP.**

3.2 Function Overloading

This is a program that shows how functions with the same name can be overloaded with different data types and different number of arguments. The function selected is based on the number and types of the formal parameters only. The type of the return value is not significant in overload resolution.

The program given below uses member functions to display words and characters. These member functions are overloaded to take arguments of type integer, char or pointer to a character string.

1. **Create a new file.**

2. **Type the following code in the 'Edit' window.**

```
#include<iostream.h>
#include<conio.h>
#include<string.h>

class Words
{
 private:
     int num;
     char title[20];
 public:
   Words()                          //constructor
   {
     num=0;
     strcpy(title,"");
   }
   void display(int i)              //display with int argument
   {
     num = i;
     cout<<"Parameter is "<<num<<endl;
     cout<<"This is not a string"<<endl;
   }
   void display(char c)             //display with char argument
   {
     title[0]=c;
     cout<<"Parameter is "<<title[0]<<endl;
     cout<<"This is a single letter"<<endl;
   }
   void display(char* s)            //display with string argument
   {
     strcpy(title,s);
      cout<<"Parameter is "<<title<<endl;
      cout<<"String length is = "<<strlen(title)<<endl;
   }
```


Durr-e-Danish Library

Scanned by CamScanner

```
    void display(char* s,int i)   //display with string and int
    {
       strcpy(title,s);
       num = i;
       cout<<"Parameters are "<<title<<" and "<<num<<endl;
       cout<<"Length of the string is = "<<strlen(title)<<endl;
    }
};
void main()
{
    clrscr();
    Words obj;

    obj.display(120);
    obj.display('B');
    obj.display("Multiple");
    obj.display("Jack Rabbit",5);
}
```

3. **Save the file as OVLD.CPP.**

4. **Build the file to get OVLD.EXE.**

5. **Execute the program, OVLD.**

OUTPUT:

```
Parameter is 120
This is not a string
Parameter is B
This is a single letter
Parameter is Multiple
String length = 8
Parameters are Jack Rabbit and 5
Length of the string is = 11
```

In this program the class `Words` is defined with a constructor and an overloaded member function called `display()`. In the `main()` program one object, `obj`, invokes the member function with different types and number of arguments.

While calling the overloaded function `display()`, the number and type of arguments is clearly used to select the proper function. In the following statements, you will notice that there is one function call with a single integer, another with a single char type variable, and yet another call with a string. The system is able to select the correct function.

```
obj.display(120);
obj.display('B');
obj.display("Multiple");
```


Durr-e-Danish Library

Scanned by CamScanner

There is another overloaded function for `display()` that takes two arguments. In the following statement it is this function that is called.

```
obj.display("Jack Rabbit",5);
```

6. Close the file, OVLD.CPP.

3.3 Friend Functions

A function can be defined as a friend function of the class to freely access the private members of the class. Neither a constructor nor a destructor can be a friend function.

In the program below we will use a friend function that adds the value assigned to two float variables belonging to two different classes called Alpha and Beta. The friend function forms a kind of bridge between the two classes and has access to the private data of both the classes.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>
#include<conio.h>
class Beta;      //needed for forward declaration

class Alpha
{
 private:
      float energy_level;
 public:
      Alpha(){ energy_level = 5.0;}        //constructor
      friend float init(Alpha, Beta);      //friend function
 };

class Beta
{
 private:
      float energy_level;
 public:
      Beta(){ene gy_level = 10.0;}         //constructor
      friend float init(Alpha, Beta);      //friend function
 };

//definition of friend function
float init(Alpha aa, Beta bb)
{
    return (aa.energy_level + bb.energy_level);
}
void main()
{
  clrscr();
```


Durr-e-Danish Library
Scanned by CamScanner

```
  Alpha obj1;
  Beta obj2;

  cout <<init(obj1,obj2)<< " is the combined energy level\n";

}
```

3. **Save the file as FRIEND.CPP.**

4. **Build the file to get FRIEND.EXE.**

5. **Execute the program, FRIEND.**

OUTPUT:

```
15.0 is the combined energy level
```

In the program above we have used two classes `Alpha` and `Beta`. Both the classes have constructors that initialise a data member `energy_level` to the values 5.0 and 10.0 in Alpha and Beta respectively. Right after the header files are included you will see that the class `Beta` has been declared as

```
class Beta;
```

The declaration is essential because the class `Beta` is referred to in the declaration of the friend function `init()` in the class `Alpha`. If we do not make this declaration the compiler will give an error since the `Beta` class specifier comes after that of `Alpha`.

The friend function `init()` takes two arguments, one of type Alpha and the other of type Beta and returns a value of type `float`. The function can access the private data member of both the classes. It adds the values of the variable `energy_level` belonging to `Alpha` and `Beta` and returns the result to the `main()` program.

A friend function does not belong to either of the classes. It is not a member function. Therefore, invoking it does not require any object name to be prefixed. The calling function `init(obj1,obj2)` in `main()` does not use any object name with a dot operator since this is a friend function only. The objects, which the friend function has to access, have to be passed to it as arguments.

- Note that though the data member in both class Alpha and Beta is called `energy_level` the scope of the variable is limited to the class in which it is declared.

- Note that the friend function is defined outside of any class. It does not need a scope resolution operator as it does not belong to either class.

6. **Close the file, FRIEND.CPP.**

Durr-e-Danish Library

Scanned by CamScanner

The Next 1 Hour:

Try out the following exercises:

1. Write a program that uses a constructor which contains a counter that is incremented every time an object is created. At the end display the result shown in the counter. Also make use of a destructor function.

2. Write a program with a Name class that can store any name in three parts such as first name, middle name and surname. Use overloaded functions to display the full name in any of several different formats such as the following;

   ```
   John Paul Doe
   J. P. Doe
   Doe, John Paul
   ```
   and any other formats you desire.

3. Write a program that uses two classes Circle and Rectangle. Each of these classes will contain data members and member functions that will be used to calculate the area of the object. For example the Circle will need a radius or diameter variable and the Rectangle will need the length and width. In addition the classes will have data members that will hold the value of the area of the objects created. Use a friend function that will access the areas of both classes, add them and display the result. You can use the following class structures:

```
class Circle{
   private:
     int radius;
     float area;
   public:
     calculate_area();
     friend display(Circle, Rectangle);
};
class Rectangle{
   private:
     int length;
     int width;
     int area;
   public:
     calculate_area();
     friend display(Circle, Rectangle);
};
```

   You can also include constructors for the classes, to create objects.

Dûrr-e-Danish Library
Scanned by CamScanner

This page has been intentionally left blank

Durr-e-Danish Library

Scanned by CamScanner

Session 4

Session Objectives

At the end of this session, the student will be able to -

- *Write programs that use*
  - *Overloaded Operators*
    - *Unary operators*
    - *Binary operators*
  - *Overloaded Assignment Operator*
  - *Copy Constructor*
  - *Type conversion of user-defined types*

The steps given in the session are detailed, comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

The First 1 Hour:

4.1 Overloaded Operators

Operators such as ++, --, <=, >, += can be overloaded to give additional operations for user-defined types, such as classes. Operator overloading can be done for unary and binary operators. We shall look at examples for both.

4.1.1 Unary Operators

In the program below we will see how the unary operators, such as the increment (++) and decrement operators (--), can be overloaded. The program will be used to convert a string with alphabets in the lower case to upper case with an overloaded operator ++. The decrement operator will be used to convert a string, which contains lower case alphabets, to upper case.

1. **Create a new file.**
2. **Type the following code in the 'Edit' window.**

```
#include<iostream.h>
#include<string.h>
#include<conio.h>

class Converter
{
  private:
     char str[80];
  public:
     Converter() {strcpy(str,"");}
     Converter(char *s) {strcpy(str,s);}
     void display() {cout<<str<<endl;}
```


Durr-e-Danish Library

Scanned by CamScanner

```
    Converter operator++()                    //prefix
    {
      return(strupr(str));
    }
    Converter operator++(int)                 //postfix
    {
      Converter ss = str;
      char *ptr = strupr(str);
      strcpy(str, ptr);
      return ss;
    }
    Converter operator--()                    //prefix
    {
     return(strlwr(str));
    }
    Converter operator--(int)                 //postfix
    {
      Converter ss = str;
      char *ptr = strlwr(str);
      strcpy(str, ptr);
      return ss;
    }
};
void main()
{
  clrscr();
  Converter s1 ="changed to UPPERCASE";
  Converter s2 ="BACK TO LOWER CASE";
  Converter s3 ="that is all for now";
  Converter s4 ="ENDING ON A LOW NOTE";
  Converter s5;

  int i,j;
  ++s1;
  cout<<"++s1 = ";
  s1.display();
  --s2;
  cout<<"--s2 = ";
  s2.display();
  s5 = s3++;
  cout<<"Result of s5 = s3++"<<endl<<"s3 = ";
  s3.display();
  cout<<"s5 = ";
  s5.display();
  s4--;
  cout<<"s4-- = ";
  s4.display();
}
```

3. Save the file as OPOVLD1.CPP.

Durr-e-Danish Library
Scanned by CamScanner

4. **Build the file to get OPOVLD1.EXE.**

5. **Execute the program, OPOVLD1.**

OUTPUT:

```
++s1 = CHANGED TO UPPERCASE
--s2 = back to lower case
Result of s5 = s3++
s3 = THAT IS ALL FOR NOW
s5 = that is all for now
s4-- = ending on a low note
```

In the above program a class named `Converter` is declared. It contains one private data member, which is a character array. Since we are using standard string functions the header file `string.h` has been included in the program.

The operator ++ is overloaded so that a string, which has alphabets in the lower case, will be converted to a string with upper case alphabets. We make use of the standard string function `strupr()` to do the conversion. Similarly, the -- operator is overloaded to convert strings from upper case to lower case. It uses the standard string function `strlwr()` to convert the string.

When ++ and -- are overloaded, the compiler does not distinguish between the prefix and postfix notation. You will notice that both the operators have two functions each. The function that uses the syntax,

```
operator++();          //prefix operation
```

does the prefix operation and the function that uses

```
operator++(int);       //postfix operation
```

performs the postfix operation. The `int` argument is just a dummy and is only included for the compiler to distinguish the two forms. We have used both notations in the program to illustrate how the operator functions work.

When we use the ++ and -- operator in statements like,

```
++s1;
--s2;
s4--;
```

it does not matter whether the operator is postfix or prefix. However, take a look at the results of the statement,

```
s5 = s3++;
```

In the output you can see that the assignment to `s5` has been done first and then the string in `s3` has been changed to upper case. This is reflected in the output as,

Durr-e-Danish Library

Scanned by CamScanner

```
s3 = THAT IS ALL FOR NOW
s5 = that is all for now
```

6. Close the file, OPOVLD1.CPP.

4.1.2 Binary Operators

Binary operators can also be overloaded as easily as unary operators. Binary operators work with two operands. In an overloaded binary operator function the operand on the right of the operator is passed to function as the argument. The left-hand side operand is accessed directly since this is the object that invokes the function.

In the program below we will overload a compound assignment operator +=. This operator combines assignment and addition in one step. The operator function that we will use will add one phrase, s2, to another,s1, and assign the result to s1. The function can also be used to add phrases in the form s3 = s1+=s2.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
const SIZE = 80;
class Phrase
{
 private:
      char str[SIZE];
 public:
      Phrase(){strcpy(str,"");}                    //constructors
      Phrase(char *s){strcpy(str,s);}
      void display(){cout<<str<<endl;}
    Phrase operator +=(Phrase aa)                  //overloaded +=
    {
    if(strlen(str)+strlen(aa.str)<SIZE)
    {
      strcat(str,aa.str);
      return(*this);
    }
      else
    {
      cout<<"String is too long";
      return(*this);
    }
  }
};
void main()
{
  clrscr();
  Phrase s1;
```

Durr-e-Danish Library

Scanned by CamScanner

```
  Phrase s2 = "Check the program";
  s1 += s2;
  s1.display();

  s1 = "Next ";
  Phrase s3;
  s3 = s1+=s2;
  s3.display();

  Phrase s4;
  Phrase s5 =" and again ";
  s4 = s2+=s5+=s5;
  s4.display();
}
```

3. **Save the file as OPOVLD2.CPP.**

4. **Build the file to get OPOVLD2.EXE.**

5. **Execute the program, OPOVLD2.**

OUTPUT:

```
Check the program
Next Check the program
Check the program and again and again
```

The program has a class called Phrase, which has one character array as a private data member. Since we are using standard string functions the header file `string.h` has been included in the program. After the header files have been included a `const` variable `SIZE` has been assigned a value of 80. This sets the maximum size for the character array in the class. While concatenating strings the length of the string is first checked to see that it does not exceed `SIZE`.

If you want to overload the += operator to perform a simple operation such as,

```
s1 += s2;
```

the function need not return a value. In such a case, the result of the operation is assigned to the operand on the left-hand side, which is the object that actually invokes the function. However, in this program we also want to be able to use the operator in more complex expressions like,

```
s3 = s1+=s2;
s4 = s2+=s5+=s5;
```

To return a value from the overloaded operator function we have used a `this` pointer. This is a more elegant usage than creating a temporary object and passing its value.

6. **Close the file, OPOVLD2.CPP.**

Durr-e-Danish Library

Scanned by CamScanner

Using the assignment operator and copy constructor we can create a `String` class. When we use the overloaded assignment operator to assign one `String` object to another we copy the string from the source into the destination object. Now the same string exists in two places in memory. While using pointers we must ensure that we are deleting the correct `String` object in a destructor function so that there are no pointers pointing to memory that has been freed.

In the program shown below we will use the `String` class and provide the full complement of operations such as constructors, copy constructor, overloaded assignment operator, and destructor.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class String
{
  private:
      char* str;
  public:
      String(char* s="")                    //one-arg constructor
      {
            cout<<"In constructor\n";
            int size = strlen(s);
            str = new char[size+1];
            strcpy(str,s);
      }
      String(String& ss)                    //copy constructor
      {
            cout<<"In copy constructor\n";
            str=new char [strlen(ss.str)+1];
            strcpy(str,ss.str);
      }
      ~String()                             //destructor
      {delete str;}
      String& operator=(String& ss)         //assignment operator
      {
            cout<<"Assignment invoked\n";
            delete str;
            str=new char[strlen(ss.str)+1];
            strcpy(str,ss.str);
            return *this;
      }
     void showstring()
     {cout<<str<<endl;}
};
```


Durr-e-Danish Library
Scanned by CamScanner

```
void main()
{
    clrscr();
    String s1 = "Strings in memory";
    cout<<"s1 = ";
    s1.showstring();
    String s2(s1),s3;
    s3=s1;
    cout<<"s2 = ";
    s2.showstring();
    cout<<"s3 = ";
    s3.showstring();
    String s4 = s1; // uses copy constructor
    cout<<"s4 = ";
    s4.showstring();
}
```

3. **Save the file as COPYCON.CPP.**

4. **Build the file to get COPYCON.EXE.**

5. **Execute the program, COPYCON.**

OUTPUT:

```
In constructor
s1 = Strings in memory
In copy constructor
In constructor
Assignment invoked
s2 = Strings in memory
s3 = Strings in memory
In copy constructor
s4 = Strings in memory
```

The class `String` that is used in this program has one private data member that is a pointer to `char`. The constructor in the class allocates memory space for a new string and copies the string into this area. The `String s1` is initialised in this manner.

We have defined another `String s2` and initialised it to `s1`. This invokes the copy constructor. Remember that a copy constructor creates a new object and copies its argument into the new object. In this case the object `s1` is copied to the object `s2`.

Next we have defined a `String s3` object. This invokes the constructor. Next we have set `s3` equal to `s1`. This invokes the assignment operator. In the assignment operator the old `String` object pointed to by `str` is first deleted and then the assignment is done.

When the following statement is executed we define and initialise an object at the same time.

```
String s4 = s1;
```


Durr-e-Danish Library
Scanned by CamScanner

We can see in the output that the copy constructor is invoked. Therefore, it is equivalent to using the syntax,

```
String s4(s1);
```

6. Close the file, COPYCON.CPP.

4.3 Type Conversion of User-Defined Types

Type conversion from basic types to user-defined types, such as objects, and vice versa, or between user-defined types have to be defined explicitly since the compiler knows only about converting basic types. In the program below we will see how conversion of an object of a class to character string is done. The conversion function overloads the typecast operator.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
const SIZE = 80;
class String
{
 private:
      char str[SIZE];
 public:
      String() {strcpy(str,"");}
      String(char *s) {strcpy(str,s);}
      void getstr()
      {
       cout<<"Enter a string: ";
       cin>>str;
      }
      void display()
      {
       cout<<"String in object = "<<str<<endl;
      }
      operator char*()
      {
         return(str);
      }
};



void main()
{
  clrscr();
```

Durr-e-Danish Library

Scanned by CamScanner

```
  String s1;
  char* p;
  s1.getstr();
  s1.display();
  p = s1;
  cout<<"String p = "<<p<<endl;
  char n[15] = "New String";
  s1 = n;
  s1.display();
}
```

3. **Save the file as CONV.CPP.**

4. **Build the file to get CONV.EXE.**

5. **Execute the program, CONV.**

You will be prompted to input a string as shown below.

OUTPUT:

```
Enter a string: Program
String in object = Program
String p = Program
String in object = New String
```

The class `String`, contains a constructor and two other member functions to accept and display the private data member, `str`, which is a character array. It also contains a conversion function to convert the object to a character pointer. In `main()` the statement

```
p = s1;
```

assigns an object to a character pointer. When the compiler sees that the right-hand side of the assignment operator contains the object and the left-hand side contains a character pointer it uses the conversion function. Without the conversion function it would not be possible to assign a user-defined type, such as an object, to a basic type.

When we assign a string `n` to the object `s1`, the compiler automatically calls the constructor that takes one argument. If a constructor was not defined then the compiler would have given an error.

6. **Close the file, CONV.CPP.**

Durr-e-Danish Library

Scanned by CamScanner

The Next 1 Hour:

Try out the following exercises:

1. Write a program that uses the Time class that was defined in the earlier session. Overload the + operator to add two Time objects instead of the addit() function used in TIME2.CPP. The program should also include operator functions for ++ and --, to increment a Time object by one second and decrement, respectively. Make sure that while incrementing and decrementing the time you check for the end of a minute, hour and day. For example, given the time 23:59:59 the result displayed after incrementing should show 00:00:00.

2. Rewrite the class String used in the program COPYCON.CPP to include functions to overload the comparison operators ==, !=, >, < so that users will be able to compare strings. Write a program to test this class.

   Hint: Use the standard `strcmp()` function from `string.h` to compare two strings s1 and s2. This function returns a negative value if one string s1 is lexicographically less than another string s2, zero if s1 is equal to s2, and a positive value if s1 is lexicographically greater than s2.

Durr-e-Danish Library

Scanned by CamScanner

Session 5

Session Objectives

At the end of this session, the student will be able to -

- *Write programs that use*
  - *Single Inheritance*
  - *Multiple Inheritance*

The steps given in the session are detailed, comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

The First 1 Hour:

5.1 Single Inheritance

Using inheritance in object-oriented programming we are able to reuse functions from one class to another. The accessibility of the base class members varies depending on whether they are private, public, or protected. Members of the derived class can access only the public and protected members of the base class.

In the program below we will see how single inheritance is implemented in a program. We will use protected data members in the base class and publicly derive two classes from a base class. We will see how the protected data members of the base class are accessed from the derived class.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>
#include<conio.h>

class vehicle
{
 protected:
   int wheels;
   double weight;
 public:
   void initialise(int whls, double wght);
   int get_wheels()          {return wheels;}
   double get_weight()       {return weight;}
   double wheel_loading()    {return weight/wheels;}
};

class car : public vehicle
{
 private:
```


Durr-e-Danish Library

Scanned by CamScanner

```
   int passenger_load;
 public:
   void initialise(int whls, double wght, int people = 4);
   int passengers()          {return passenger_load;}
};

class truck : public vehicle
{
 private:
   int passenger_load;
   double payload;
 public:
   void init_truck(int number = 2, double max_load = 24000.0);
   double efficiency();
   int passengers()          {return passenger_load;}
};

// initialise to any data desired
void vehicle::initialise(int whls, double wght)
{
   wheels = whls;
   weight = wght;
}

void car::initialise(int whls, double wght, int people)
{
   passenger_load = people;
   wheels = whls;
   weight = wght;
}

void truck::init_truck(int number, double max_load)
{
   passenger_load = number;
   payload = max_load;
}

double truck::efficiency()
{
   return payload / (payload + weight);
}

void main()
{
clrscr();
vehicle bicycle;

   bicycle.initialise(2, 25);
   cout << "The bicycle has " <<
   bicycle.get_wheels() << " wheels.\n";
   cout << "The bicycle's wheel loading is "
   <<bicycle.wheel_loading()<<" pounds on a single tire.\n";
```


Durr-e-Danish Library
Scanned by CamScanner

```
   cout << "The bicycle weighs "
   <<bicycle.get_weight()<<" pounds.\n\n";

car coupe;

   coupe.initialise(4, 3500.0, 5);
   cout<<"The coupe carries "<<coupe.passengers()
   <<" passengers.\n";
   cout<<"The coupe weighs "<<coupe.get_weight()<<" pounds.\n";
   cout<<"The coupe's wheel loading is "
   <<coupe.wheel_loading()<<" pounds per tire.\n\n";

truck van;

   van.initialise(18, 12500.0);
   van.init_truck(1, 33675.0);
   cout<<"The van weighs "<<van.get_weight()<< " pounds.\n";
   cout<<"The van's efficiency is "
   <<100.0 * van.efficiency() << " percent.\n";
}
```

3. **Save the file as SINGLE.CPP.**

4. **Build the file to get SINGLE.EXE.**

5. **Execute the program, SINGLE.**

OUTPUT:

```
The bicycle has 2 wheels.
The bicycle's wheel loading is 12.5 pounds on a single tire.
The bicycle weighs 25 pounds.

The coupe carries 5 passengers.
The coupe weighs 3500 pounds.
The coupe's wheel loading is 875 pounds per tire.

The van weighs 12500 pounds.
The van's efficiency is 72.929072 percent.
```

In the above program a class named `vehicle` is declared. It contains two protected data members, and several functions in the public section. The class `vehicle` is the base class for two derived classes, `car` and `truck`.

Both the classes vehicle and car have a function called `initialise()`. You can see that in the `initialise()` function for the class `car` the protected data members of the base class, namely, `wheels` and `weight`, are accessed and assigned values. This function also initialises the data member of the class `car`.

Durr-e-Danish Library

Scanned by CamScanner

Another function that accesses the protected data member of the base class is the function `efficiency()` of class `truck`. The function uses the weight data member of the base class to calculate the efficiency of the `truck`.

In `main()` we have defined one object of the type `vehicle` and one object each of the derived classes. You can see how the derived class objects invoke the base class member functions as in,

```
coupe.get_weight()
coupe.wheel_loading()
van.get_weight()
```

The class `truck` does not have an `initialise()` function. Therefore when we use the statement,

```
van.initialise(18, 12500.0);
```

where `van` is an object of type `truck`, the function call invokes the `initialise()` function of the base class.

6. Close the file, SINGLE.CPP.

5.2 Multiple Inheritance

Multiple inheritance occurs when a class derives from more than one base class. It inherits the properties of its base classes. A derived class inherits data members and function from all its base classes regardless of whether the inheritance is public, private, or protected. When the base classes have member functions with identical names we need to use the scope resolution operator to refer to the correct class.

In the program shown below we will use two base classes from which we have one publicly derived class. The program will take information from the user regarding a student's personal information and academic record and display it.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>
#include<iomanip.h>
#include<conio.h>
class person_data          //one base class
{
private:
      char name[25];
      int roll_no;
      char sex;
public:
      void getinfo();
      void display();
};
```

Durr-e-Danish Library

Scanned by CamScanner

```
class academics                    //second base class
{
private:
    char course_name[25];
    int semester;
    char grade[3];
public:
    void getinfo();
    void display();
};
                                   //public derived class
class stud_scholarship: public person_data, public academics
{
private:
    float amount;
public:
    void getinfo();
    void display();
};

void person_data::getinfo()
{
  cout<<"\nName? ";
  cin>>name;
  cout<<"Roll number? ";
  cin>>roll_no;
  cout<<"Sex (F/M)? ";
  cin>>sex;
}
void person_data::display()
{
 cout<<name<<"\t";
 cout<<roll_no<<"\t";
 cout<<sex<<"\t";
}
void academics::getinfo()
{
  cout<<"Course name (BA/MBA/MCA etc)? ";
  cin>>course_name;
  cout<<"Semester(1/2/3/...)? ";
  cin>>semester;
  cout<<"Grade (A,B,B+,B-...)? ";
  cin>>grade;
}
void academics::display()
{
  cout<<course_name<<"\t";
  cout<<semester<<"\t";
  cout<<grade<<"\t";
}
void stud_scholarship::getinfo()
{
```


Durr-e-Danish Library

Scanned by CamScanner

```
  person_data::getinfo();
  academics::getinfo();
  cout<<"Assistance received (in $)? ";
  cin>>amount;
}
void stud_scholarship::display()
{
  person_data::display();
  academics::display();
  cout<<setprecision(2);
  cout<<amount<<endl;
}
void main()
{
  clrscr();
  stud_scholarship  obj;
  cout<<"Please enter the following information:";
  obj.getinfo();
  cout<<endl;
  cout<<"Name    Rollno     Sex    Course   Semester  Grade";
  cout<<"  Amount\n";
  obj.display();
}
```

3. **Save the file as MULTI.CPP.**
4. **Build the file to get MULTI.EXE.**
5. **Execute the program, MULTI.**

You will be prompted to enter some information as given below.

OUTPUT:

```
Please enter the following information:
Name? Jack
Rollno? 12
Sex (F/M)? M
Course name (BA/MBA/MCA etc)? MCA
Semester(1/2/3/...)? 2
Grade (A,B,B+,B-...)? A
Assistance received (in $)? 234.56

Name  Rollno     Sex   Course  Semester  Grade  Amount
Jack  12         M      MCA      2        A      234.56
```

The base class `person_data` that is used in this program contains the data members' `name`, `roll_no` and `sex`. Another base class, `academics`, contains the data members, `course_name`, `semester` and `grade`. The derived class `stud_scholarship` contains the data member `amount`. The derived class has been declared as publicly inherited. To be able

Durr-e-Danish Library

Scanned by CamScanner

to access the private data members of its base classes, the derived class calls the member functions of the base classes.

Since both the base classes have functions `getinfo()` and `display()` in the derived class the functions are called using the scope resolution operator as given in the statements.

```
person_data::getinfo();
academics::getinfo();
```

6. Close the file, MULTI.CPP.

5.3 Constructors under inheritance

When constructors are used in the base class and derived classes, the base class is initialised before the derived class, using either the default constructor or a constructor with arguments depending on the code of the constructor of the derived class. The base class constructors are invoked first. The order in which they are invoked is the same order in which the base classes are declared in the derived class declaration. If there are any member objects in the class, they are initialised next, in the order they appear in the class declaration. Finally the derived class constructor is called.

In this program we will see how the constructors are called when multiple inheritance is used. The classes have all got default constructors. Let us see the order in which the constructors are called.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>
#include<conio.h>
class Alpha
{
 public:
   Alpha()
   {cout<<"constructor of class Alpha"<<endl;}
};
class Beta:public Alpha
{
 public:
   Beta():Alpha()
   {cout<<"constructor of class Beta, derived from Alpha"<<endl;}
};
class Gamma:public Alpha
{
 public:
   Gamma():Alpha()
  {cout<<"constructor of class Gamma, derived from Alpha"<<endl;}
};
class Delta:public Gamma,public Beta
```


Durr-e-Danish Library

Scanned by CamScanner

```
{
 Beta b;          //base class object
 public:
    Delta(): Gamma(),Beta()
    {cout<<"constructor of class Delta, derived from Beta and "
       <<"Gamma,\n"<<"having a Beta object inside"<<endl;}
};

void main()
{
 clrscr();
 cout<<"Defining an object of class Alpha\n";
 Alpha aa;
 cout<<endl;

 cout<<"Defining an object of class Beta\n";
 Beta bb;
 cout<<endl;

 cout<<"Defining an object of class Gamma\n";
 Gamma cc;
 cout<<endl;

 cout<<"Defining an object of class Delta\n";
 Delta dd;
 cout<<endl;
}
```

3. **Save the file as MULTCON.CPP.**

4. **Build the file to get MULTCON.EXE.**

5. **Execute the program, MULTCON.**

OUTPUT:

```
Defining an object of class Alpha
constructor of class Alpha

Defining an object of class Beta
constructor of class Alpha
constructor of class Beta, derived from Alpha

Defining an object of class Gamma
constructor of class Alpha
constructor of class Gamma, derived from Alpha

Defining an object of class Delta
constructor of class Alpha
constructor of class Gamma, derived from Alpha
constructor of class Alpha
```


Durr-e-Danish Library

Scanned by CamScanner

```
constructor of class Beta, derived from Alpha
constructor of class Alpha
constructor of class Beta, derived from Alpha
constructor of class Delta, derived from Beta and Gamma
having a Beta object inside
```

In the above program we defined four classes. There is one base class `Alpha` from which two classes, `Beta` and `Gamma`, are directly derived. For the class `Delta`, the class `Alpha` is an indirect base class, and the classes `Beta` and `Gamma` are direct base classes. The class `Delta` also has a data member, which is an object of the `Beta` class.

Study the output carefully to see the order in which the constructors are called. The output generated for the constructors of the objects of type `Alpha`, `Beta` and `Gamma` is straightforward.

When we come to the object of type `Delta` we can see that the constructor of `Alpha` is called first. In fact it is called every time the constructors of `Gamma` and `Beta` are called. You will see that the constructors of base classes `Gamma` and `Beta` appear in the order in which they are declared in the declaration of class `Delta`.

```
class Delta:public Gamma,public Beta
```

The `Gamma` class constructor is called first and then that for `Beta`. After the constructors of `Gamma` and `Beta` are called we see that the constructor of `Beta` is called again. Since there is an object of type `Beta` in the class `Delta`, that object is initialised before the code of the `Delta` class constructor is generated.

6. Close the file, MULTCON.CPP.

Durr-e-Danish Library

Scanned by CamScanner

The Next 1 Hour:

Try out the following exercises:

1. Rewrite the program MULTCON.CPP used in this Session with a data member of type integer for each of the classes Alpha, Beta, Gamma, and Delta. Provide constructors with one argument to initialise this integer type data member. Also provide destructors for the classes and output the string "In destructor of class Class_name" where Class_name is the name of the class. Observe the way in which the constructors and destructors are called.

2. In a research lab a small catalogue is maintained. The catalogue includes information on articles (published in journals) and books. Create a base class for the `catalogue` in which the private data members are title, author name and a date. There should also be member functions `getdata()` and `display()` to get and display the data members respectively. The class articles and class books should be derived classes. Additional data members for the class of `articles` must include journal name, and a volume number. For the class of `books`, provide additional data members for the name of the publisher, and the city where the book was published. These derived classes should also have member functions `getdata()` and `display()` to get information from the user and to display it. Write a program to test the classes.

Durr-e-Danish Library

Scanned by CamScanner

Session 6

Session Objectives

At the end of this session, the student will be able to -

- *Write programs that use Polymorphism*
  - *Virtual functions*
  - *Pure Virtual Functions*
  - *Virtual destructors*

The steps given in the session are detailed, comprehensive and carefully thought through. This has been done so that the learning objectives are met and the understanding of the tool is complete. Please follow the steps carefully.

The First 1 Hour:

6.1 Polymorphism

In the context of programming, polymorphism allows you to refer to objects of different classes by means of the same program item and to perform the same operation in different ways, depending on which object is being referred to. In C++, one way polymorphism is implemented is through the use of virtual functions. When virtual functions are used, the system makes the decision of which method to actually call based on which object the pointer is pointing to. The decision of which function to call is not made during the time when the code is compiled but when the code is executed. This is dynamic binding and can be very useful in some programming situations.

6.1.1 Virtual Functions

A virtual function must have an implementation available in the base class, which will be used if one is not available in one or more of the derived classes. In the program below we will see how virtual functions are used. We will use a protected data member in the base class and publicly derive two classes from the base class. The base class will have a virtual member function that is redefined in the derived classes.

1. **Create a new file.**
2. **Type the following code in the 'Edit' window.**

```
#include<iostream.h>
#include<string.h>
#include<conio.h>
class person
{
 protected:
     char* name;
 public:
     person(char* n)
     {
```

Durr-e-Danish Library

Scanned by CamScanner

```
        name = n;
       }
       virtual void print()
       {
        cout<<"My name is "<<name<<endl;
       }
 };
class foreigner: public person
{
public:
       foreigner(char* n):person(n){ }
       void print()
       {
        cout<<"ll mio nome e "<<name<<endl;
       }
};
class alien: public person
{
 public:
       alien(char* s):person(s){ }
       void print()
       {
         cout<<"##$&(^@$%@!#@$%~~***@##"<<name<<endl;
         cout<<"Sorry, there is a communication problem"<<endl;
       }
};

void main()
{
  clrscr();
  person* person1;
  person* person2;

  person1 = new person("Jack");
  person2 = new foreigner("Maria");

  cout<<"Introducing a Person:"<<endl;
  person1->print();

  cout<<"Introducing a Foreigner:"<<endl;
  person2->print();

  person1 = new foreigner("Al Pacino");
  cout<<"Reintroducing the Person:"<<endl;
  person1->print();

  person* person3;
  person3 = new alien("Martian");
  cout<<"Introducing the alien:"<<endl;
  person3->print();
}
```

Durr-e-Danish Library

Scanned by CamScanner

3. **Save the file as VIRTUAL1.CPP.**

4. **Build the file to get VIRTUAL1.EXE.**

5. **Execute the program, VIRTUAL1.**

OUTPUT:

```
Introducing the Person:
My name is Jack
Introducing the Foreigner:
Il mio nome e Maria
Reintroducing the Person:
Il mio nome e Al Pacino
Introducing the alien:
##$&(^@$%@!#@$%~~***@##Martian
Sorry, there is a communication problem
```

In the above program a class named `person` is declared. It contains a data member that holds the name of a `person` and a function to print that name. The class `person` is the base class for two publicly derived classes, `foreigner` and `alien`.

Both the derived classes have `print()` functions like the base class except in the contents that they print out. In the class `person` the definition of the function is preceded by the keyword virtual which lets the compiler use dynamic binding for the function.

In the program, initially there are two pointers of type `person`. One pointer of type `person` is associated with an object of the base class by using the statements,

```
person* person1;
person1 = new person("Jack");
```

Another pointer of type `person` is associated with an object of the derived class `foreigner`, as seen in the statements,

```
person* person2;
person2 = new foreigner("Maria");
```

Thus we are able to handle objects of class `person` and `foreigner`. The objects are created using the new operator and initialised. The function call

```
person2->print();
```

used to invoke a member function with a pointer of type `person` does not call the function

```
person::print()
```

but calls the function

Durr-e-Danish Library

Scanned by CamScanner

```
foreigner::print()
```

The compiler decides which of the two functions is to be called only at run time and the choice is based on the value assigned to the pointer `person2`.

In the beginning of `main()`, the pointer `person1` was associated with an object of class `person`, so the compiler calls the function of this class. On the other hand, the pointer `person2`, was associated at run-time with an object of class `foreigner`, so the compiler calls the function of this class.

In the next part of the program, the pointer `person1` is associated with an object of the derived class. Therefore the statement,

```
person1->print();
```

invokes the function of the derived class. In a similar manner the pointer `person3` is associated with an object of the derived class `alien` and the `print()` function of that class is called.

The number and types of the parameters must be identical for all the `print()` functions, since a single statement can be used to call any of them.

6. Close the file, VIRTUAL1.CPP.

6.1.2 More Virtual functions

Let us look at another program that makes use of virtual functions. In the program shown below we will use one base class from which we have one publicly derived class. The program is an example of how text can be generated in a desired format. The derived class overrides only certain functions of the base class.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>
#include<conio.h>

class Format
{
public:
   void display_form();
   virtual void header() { cout << "This is a header\n"; }
   virtual void body()   { cout << " This is body text\n"; }
   virtual void footer() { cout << "This is a footer\n\n"; }
};
void Format::display_form()
{
   header();
   for (int index = 0 ; index < 3 ; index++)
   {
```


Durr-e-Danish Library
Scanned by CamScanner

```
      body();
   }
   footer();
}
//This class overrides two of the virtual methods of the base
//class
class MyForm: public Format
{
   void header() { cout << "This is the new header\n"; }
   void footer() { cout << "This is the new footer\n"; }
};
void main()
{
   clrscr();
   Format* first_form = new Format;
   first_form->display_form();    // A call to the base class

   delete first_form;
   first_form = new MyForm;
   first_form->display_form();    // A call to the derived class
}
```

3. **Save the file as VIRTUAL2.CPP.**
4. **Build the file to get VIRTUAL2.EXE.**
5. **Execute the program, VIRTUAL2.**

OUTPUT:

```
This is a header
 This is body text
 This is body text
 This is body text
This is a footer

This is the new header
 This is body text
 This is body text
 This is body text
This is the new footer
```

The base class `Format` that is used in this program contains no data members but four member functions that output a header, body and footer. The body text is generated three times. The derived class overrides two of the functions, which generate the header and footer. It is possible that some time later you may use a derived class which also overrides the `body()` function. Therefore we have made that function also virtual.

Durr-e-Danish Library
Scanned by CamScanner

In `main()` we first create a pointer to an object of the type `Format` which is the base class. The first call to `display_form()` uses the functions relevant to the base class. This can be seen in the output generated.

The pointer `first_form` is next associated with an object of the derived class. Now when `display_form()` is called, the functions `header()` and `footer()` of the derived class are called.

6. Close the file, VIRTUAL2.CPP.

6.2 Pure Virtual functions

A pure virtual function is declared by assigning the value of zero to the function. A class containing one or more pure virtual functions cannot be used to define an object. The class is therefore only useful as a base class to be inherited into a useable derived class. It is called an abstract class. Every derived class must include a function for each pure virtual function that is inherited from the base class if it will be used to create an object.

In the program below we will modify the previous program so that one of the functions is made a pure virtual function. You will notice that this program uses an abstract base class which makes it illegal to use an object of the base class as we did in the last program. For that reason, some of the main program is commented out.

1. Modify the program VIRTUAL2.CPP as given below.

```
#include<iostream.h>
#include<conio.h>

class Format
{
public:
   void display_form();
   virtual void header() { cout << "This is a header\n"; }
   virtual void body() = 0;     // Pure virtual function
   virtual void footer() { cout << "This is a footer\n\n"; }
};


void Format::display_form()
{
   header();
   for (int index = 0 ; index < 3 ; index++)
   {
      body();
   }
   footer();
}

//This class overrides two of the virtual methods of the base
//class
```


Durr-e-Danish Library
Scanned by CamScanner

```
class MyForm : public Format
{
   void body() { cout << " This is the new body text\n"; }
   void footer() { cout << "This is the new footer\n"; }
};

void main()
{
   clrscr();
// The next three lines of code are invalid since Format is
// now an abstract class and an object cannot be created

// Format* first_form = new Format;
//    first_form->display_form();   // A call to the base class
//    delete first_form;

// An object of the derived class
   Format* first_form = new MyForm;
   first_form->display_form();     // A call to the derived class

}
```

2. **Save the file as VIRTUAL3.CPP.**

3. **Build the file to get VIRTUAL3.EXE.**

4. **Execute the program, VIRTUAL3.**

OUTPUT:

```
This is a header
 This is the new body text
 This is the new body text
 This is the new body text
This is the new footer
```

In the above program we have modified the `body()` function of the class `Format` and made it a pure virtual function. In the derived class a definition of the function is given. In the previous program the `body()` function had not been redefined in the derived class.

The derived class also defines the `footer()` function which overrides the definition of the base class function.

Since we have a pure virtual function in class `Format`, it is an abstract base class. The only objects that can be created are that of the derived class. You can see the difference in the output between the previous program and this one.

5. **Close the file, VIRTUAL3.CPP.**

Durr-e-Danish Library

Scanned by CamScanner

6.3 Virtual destructors

Destructors are invoked to automatically free memory that has been assigned to objects. However the destructor member function of the derived class is not invoked to free the memory storage which was allocated by the constructor of the derived class. This is because destructor functions are non-trivial and the message does not reach the destructor member function under dynamic binding. In such a situation it is better to have a virtual destructor function.

6.3.1 Base class with non-virtual destructor

In the program shown below we will use a base class which does not have a virtual destructor. You can see from the output displayed that the destructor of the derived class is not invoked.

1. Create a new file.

2. Type the following code in the 'Edit' window.

```
#include<iostream.h>
#include<string.h>
#include<conio.h>

class Sentence
{
  private:
      char* str;
  public:
      Sentence();
      ~Sentence();
};

class Phrase: public Sentence
{
  private:
      char* phr;
  public:
      Phrase();
      ~Phrase();
};
Sentence::Sentence()          //base constructor
{
  str = new char[50];
  strcpy(str,"This sentence is worth considering\n");
  cout<<str;
}
Sentence::~Sentence()         //base's destructor
{
  delete[] str;
  cout<<"Releasing the worthy sentence\n";
}
Phrase::Phrase()              //derived class constructor
{
```


Durr-e-Danish Library
Scanned by CamScanner

```
  phr = new char[25];
  strcpy(phr,"This is only a humble phrase\n");
  cout<<phr;
}
Phrase::~Phrase()              //derived class destructor
{
  delete[] phr;
  cout<<"Releasing the humble phrase\n";
}

void main()
{
  clrscr();
  Sentence* ptr = new Phrase;
  delete ptr;
}
```

3. **Save the file as VIRTUAL4.CPP.**

4. **Build the file to get VIRTUAL4.EXE.**

5. **Execute the program, VIRTUAL4.**

OUTPUT:

```
This sentence is worth considering
This is only a humble phrase
Releasing the worthy sentence
```

The base class `Sentence` that is used in this program contains one data member and a constructor and destructor. The derived class follows the same format.

In `main()` a pointer of type `Format` is associated with an object of the derived class as seen in the statement,

```
Sentence* ptr = new Phrase;
```

Now when `ptr` is deleted we see in the output that only the base class destructor is invoked. As long as the type specified in the delete operation matches the actual type of the object the compiler will supply the right destructor. Here we are deleting a pointer to the base but the object pointed to is of the derived class. This problem can be overcome by making the base class destructor virtual.

6. **Close the file, VIRTUAL4.CPP.**

6.3.2 Base class with a Virtual Destructor

In the program shown below we will modify the previous program so that the base class destructor is made virtual. It is a good idea to supply a virtual destructor in all classes that act as

Durr-e-Danish Library
Scanned by CamScanner

a base class, specially when objects of the derived class are manipulated and deleted through a pointer to the base.

1. Modify the program VIRTUAL4.CPP as given below.

```
#include<iostream.h>
#include<string.h>
#include<conio.h>

class Sentence
{
  private:
      char* str;
  public:
      Sentence();
      virtual ~Sentence();          //destructor made virtual
};

class Phrase: public Sentence
{
  private:
      char* phr;
  public:
      Phrase();
      ~Phrase();
}
Sentence::Sentence()                //base constructor
{
  str = new char[50];
  strcpy(str,"This sentence is worth considering\n");
  cout<<str;
}
Sentence::~Sentence()               //base's destructor
{
  delete[] str;
  cout<<"Releasing the worthy sentence\n";
}
Phrase::Phrase()                    //derived class constructor
{
  phr = new char[25];
  strcpy(phr,"This is only a humble phrase\n");
  cout<<phr;
}
Phrase::~Phrase()                   //derived class destructor
{
  delete[] phr;
  cout<<"Releasing the humble phrase\n";
}

void main()
{
```


Durr-e-Danish Library
Scanned by CamScanner

```
  clrscr();
  Sentence* ptr = new Phrase;
  delete ptr;
}
```

2. Save the file as VIRTUAL5.CPP.

3. Build the file to get VIRTUAL5.EXE.

4. Execute the program, VIRTUAL5.

OUTPUT:

```
This sentence is worth considering
This is only a humble phrase
Releasing the humble phrase
Releasing the worthy sentence
```

The above program is similar to the previous one except that the base class destructor has been made virtual. You can see from the output that the derived class destructor is called before that of the base class.

5. Close the file, VIRTUAL5.CPP.

The Next 1 Hour:

Try out the following exercises:

1. Use the classes catalogue, articles and books created in the previous lab session exercise. Make the functions getdata() and display() of the base class as virtual functions. Write a program to create an array of ten pointers of base class type. You should ask the user whether he wants to enter the data for a book or for an article and accordingly create an object of correct type and use the appropriate functions. Also, display the contents of all the objects.

2. Create a base class called parent which has data members that define the name, number of children and spouse's name and member function get the values and display them. From this class derive two classes child1 and child2. Both classes should have member functions that will access the data in the base class parent. Also create a class grandchild derived from child1 and child2. In this class provide a member function that will display the data from the parent class.

   a. Write a program to test these classes.
   b. Make the class parent a virtual base class and compare the output that will be generated.

Durr-e-Danish Library
Scanned by CamScanner

Durr-e-Danish Library

Scanned by CamScanner

On-Line Training Lab Guide

SESSION 1

This session is of 2 hours consisting of first one hour of surfing and a quiz, followed by second one hour to try out some exercise on the tool.

The First One Hour:

Instructions for the first One hour of On-Line Training Lab session.

Session Number	Session Contents	Name of the Home page
Session 1	Object Oriented Programming with C++	Main.htm

The objectives of this session are to learn the following:

- *Functions with Default Arguments*
- *Inline Functions*
- *The Const keyword*
- *Static Class members*
  - *Static Data Member*
  - *Static Member Function*
- *Type casting of variables*
- *Quiz*

The Next One Hour:

Try out the following exercises:

1. Write a program using the Time class of TIME.CPP used in Session 2. Provide public member functions to:

a. Set the time(hour, minute, second)
b. Set the hour
c. Set the minute
d. Set the second
e. print the time in military format (i.e. 21:12:30 (hh:mm:ss))
f. print the time in standard format (i.e. 08:43:30 A.M.)

Use constructors with default arguments to set the time.

2. Create a savings account class. Use a static data member that contains the annual interest rate for the savings accounts. The class should contain a private data member balance indicating the amount the account holder currently has in the account. Provide a member function to calculate the monthly interest. The interest is calculated by multiplying the balance with the annual interest rate divided by 12. This interest should be added to the savings balance. Provide a static member function `modifyintrate` that sets the static annual interest rate to a new value. Write a program to test the class using two savings account class objects, saver1

Durr-e-Danish Library
Scanned by CamScanner

and saver2, with balances of $2000.00 and $3500.00, respectively. Set the annual interest rate to 3%. Calculate the next month's interest and print the new balances for each of the account holders.

3. Write a program with an inline function to find the smaller of any two integer numbers that are given by the user. The function should set the smaller number to zero.

Durr-e-Danish Library

Scanned by CamScanner

On-Line Training Lab Guide

SESSION 2

This session is of 2 hours consisting of first one hour of surfing and a quiz, followed by second one hour to try out some exercise on the tool.

The First One Hour:

Instructions for the first One Hour of On-Line Training Lab session.

Session Number	Session Contents	Name of the Home page
Session 2	Object Oriented Programming with C++	Main.htm

The objectives of this session are to learn the following:

- *Concepts of Input/Output*
  - *Other istream and ostream functions*
  - *Formatted Input/Output*
- *Preprocessors*
  - *Simple macro definitions*
  - *Conditional compilation*
  - *Macros with parameters*
- *Creating and using own header files*
- *Quiz*

The Next One Hour:

Try out the following exercises:

1. Write a program to print an integer number using the `setf()` manipulator function. If the number is 110 the following output should be generated.

   showpos: [+110]
   Hex: [6e]
   Base prefix: [0x6e]
   Uppercase hex: [0X6E]
   Left justified: [+110]
   Right justified: [+110]

   In the last two lines of output set the width to 6.

2. Write a program using a macro to define the square of an integer number. Make sure that the arguments used in the macro are enclosed in parentheses or you will get peculiar results.

3. Write a header file for the `Time` class, which we have used in previous sessions. You can provide member functions to do the following:

Dürr-e-Danish Library

Scanned by CamScanner

a. set the time (hour, minute, second)
b. set the hour
c. set the minute
d. set the second
e. print the time in the format 20:12:43
f. print the time in the format 08:43:54 A.M
g. get the hour
h. get the minute
i. get the second

Put the class declaration in a file TIME.H, the definitions of the member functions in TIME1.CPP. Write a program TIME2.CPP to implement the class.

Durr-e-Danish Library

Scanned by CamScanner

A.1 Input/Output

Most languages have facilities for input and output built into the language. C and C++ do not have built-in statements for I/O. The C language has functions to achieve the I/O and C++ has classes to do the same.

A.2 Features of iostream.h

C++ views input and output as a stream of bytes. The I/O stream is a sequence of characters written to the standard output device or read from the keyboard. The standard input and output operations in C++ are normally performed by using the I/O stream using `cin` for input and `cout` for output. C++ associates a buffer to every stream. Input is taken from this buffer and output written to the buffer.

The `iostream.h` header file contains several classes to manage the I/O from and to buffers. Some of these classes are:

`streambuf`	This class provides memory for a buffer along with methods to use the buffer
`ios`	This class defines certain properties of a stream, like the mode it is opened in, read or write or both, whether it is a binary or text stream.
`ostream`	This class is derived from the `ios` class and contains methods to perform output
`istream`	This class is also derived from the `ios` class and contains methods to perform input.
`iostream`	Derived from `ostream` and `istream` classes to do both input and output.

The stream `istream` and `ostream` classes are both derived through single inheritance from the `ios` base class. The `iostream` class is derived through multiple inheritance from both the `istream` class and the `ostream` class. These inheritance relationships are shown in the figure below.

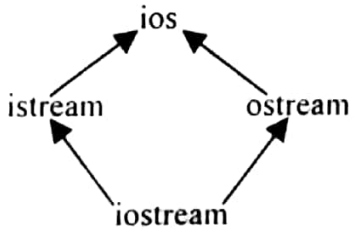


Figure 1: Input stream and output stream

On including the `iostream.h` header file, a program automatically gets the following four objects of the above classes.

`cin`	This object corresponds to the standard input stream, which by default is the keyboard.
`cout`	This object corresponds to the standard output stream, which is the video display, by default.

Durr-e-Danish Library
Scanned by CamScanner

`cerr`	The standard error stream, which is the video display, by default. It is used like the stderr device associated with library functions for posting error messages.
`clog`	This object corresponds to the standard error stream, which is the screen by default. Output to `clog` is buffered.

Outputs to objects of `cerr` are unbuffered. This means that each insertion to `cerr` causes its output to appear immediately. It is appropriate for promptly informing the user if any errors occur.

Output to `clog` is buffered. This means that each insertion to `clog` causes its output to be held in a buffer until the buffer is filled or until the buffer is flushed.

A.3 Output with cout

Output on the standard stream is done using the `cout` object of the `ostream` class. This class overloads the bit-wise left-shift operator <<. This operator is also called the insertion operator. In the following statement,

```
cout<<variable;
```

`variable` can be any basic data type. The operator function for the insertion operator is overloaded for each of the data types. If the variable in the above statement is an integer, then the function with the corresponding argument is invoked. The prototype of this function would be:

```
ostream &operator<<(int n);
```

The insertion operator recognises all the basic data types of C++. It is overloaded for all of them.

Apart from the basic data types the overloaded insertion operator also has a function whose argument defines a pointer of type `void`. This function prints the address of a pointer. For example, if you want to print the address of a string, type cast it to `void*`.

Example 1

```
char* name = "Programming";
cout<<name<<"\n";
cout<<(void*)name;
```

Output:

```
Programming
0x2ef1ffe
```

A.4 Other ostream functions

Apart form the overloaded operator<<() function, the `ostream` classes provides the `put()` and `write()` functions for output. The `put()` function has the following prototype:

```
ostream &put(char);
```


Durr-e-Danish Library

Scanned by CamScanner

This function prints only characters. It can be invoked as follows:

```
cout.put('Y');
```

This function also returns a reference to the `cout` object and hence can be used as follows:

```
cout.put('Y').put('E').put('S').put('\n');
```

The `write()` function prints an entire string and has the following prototype.

```
ostream &write(char*, int);
```

The first argument gives the address of the character string to be printed and the second specified the number of characters to be printed. This function does not recognise the null character. If the second argument is more than the length of the string, then it prints some undefined characters.

Example 2

```
char* abc = "Check";
int len = strlen(abc);

cout.write(abc,len-1);
```

Output:
```
Chec
```

A function `flush()` can be used to flush the output buffer. Another function `endl()` prints a newline character. They can be used as follows:

```
cout<<"Input a newline"<<endl;
cout<<"Output to be flushed"<<flush;
```

These functions can be used with the insertion operator and the `cout` object, because the `ostream` class has a function overloaded for them. They can also be used in the format:

```
flush(cout);
endl(cout);
```

A.5 Formatted Output with cout

The overloaded functions for the insertion operator << of the `ostream` class format the variables before displaying.

➢ Default formats

The default formats are as follows:

`char`	displayed as a character, uses one position.
`int`	displayed as a decimal integer. uses space just enough for one number and a minus sign if the number is negative.

Durr-e-Danish Library

Scanned by CamScanner

`float` and `double`	displayed with six decimal places, except for the trailing zeros. The field is just wide enough for the number and a minus sign if the number is negative. The number is in fixed point notation or normalised form, whichever takes less space.
Character strings	displayed in a field equal to the length of the string.

➢ Manipulators

Manipulator functions are special stream functions that change certain characteristics of the input and output. Using these functions you can format the input and output streams.

The following are the list of manipulators used in C++.

`hex, dec, oct`	sets the base in which a number will be printed
`setbase`	defines the base of the numeral value variable
`setw` or `width()`	used to specify the minimum number of character positions on the output field that a variable will use
`setfill` or `fill()`	used to specify a different character to fill any unused field width of a value
`setprecision` or `precision()`	used to control the number of digits of an output stream display of a floating point value
`ends`	used to attach a null terminating character ('\0') at the end of a string
`ws`	used to ignore the leading white space that precedes the first field
`flush`	used to completely empty the stream associated with the output For output on the screen, this is not necessary as all output is automatically flushed.
`setf`	used to control different output settings.

The `hex`, `dec`, `oct`, `ws`, `endl`, and `flush` manipulator function are defined in `iostream.h`. The manipulators that take arguments like `setbase`, `width()`, `fill()`, `precision()` are defined in `iomanip.h`, which must be included in any program that uses them.

➢ Changing Field widths, padding extra places and setting the precision

The manipulators that control the width of the output are called width specifiers. The `width()` or `setw()` manipulators arethe width specifiers available. The default width is always zero. That is because the `cout` object always uses space just wide enough to output the values. You can use the `width()` function to change the current width setting. The argument passed to it is the new width to be set. The `width()` function affects only the next item displayed. Afterwards the field width reverts to the default value.

By default, C++ pads spaces to extra places, if the width specified is more than the actual output. With the function `fill()` or `setfill()` you can set a new fill character.

While displaying floating point values, by default, C++ prints six decimal places after the decimal point except for the trailing zeros. The function `precision()` or `setprecision()` can be used to change this default value. The function truncates the trailing digits and rounds them up.

Durr-e-Danish Library

Scanned by CamScanner

➢ The setf() function

The `ios` class contains a function `setf()` for controlling several formatting features. This class defines a set of flags that can be used as arguments to the `setf()` function. This function has the following prototypes:

```
long setf(long flagbit);
long setf(long flagbit, long field);
```

These functions set a flag, as maintained by the `ios` class. They also return the value of the flag as it was, before setting it. The first prototype has only one argument. It can be any one of the following:

Flags	Meaning
showbase	Use base prefixes (0,0x) for output
showpoint	Show decimal point and trailing zeros
uppercase	Use uppercase letters for hex output(A-F)
showpos	Use + for positive numbers

Table 1

The flags are defined in the `ios` class definition and therefore have a class scope. They have to be prefixed by the name of the class with the scope resolution operator. For example,

```
cout.setf(ios::showpoint);
```

The second prototype for `setf()` has two arguments. The first argument defines the bit settings that have to be changed. The second argument defines what is to be changed. These arguments can be any of the following:

Flags		Meaning
Second argument	First argument	
basefield	dec	Use decimal base
	oct	Use octal base
	hex	Use hexadecimal base
floatfield	fixed	Use fixed-point notation
	scientific	Use scientific notation
adjustfield	left	Left justified output
	right	Right justified output
	internal	Left justify the sign and base prefix, but right justify the value

Table 2

All the first and second arguments should be prefixed by the name of the class. For example,

```
cout<<setf(ios::hex,ios::basefield);
cout<<1234;
Output:
0x4d2
```


Durr-e-Danish Library
Scanned by CamScanner

Once a base is set it is used till it is reset. For the `adjustfield` flag right adjustment is the default.

A.6 Input with cin

Input to a program is done through the `cin` object of `istream` class. The `cin` object overloads the right shift operator >> for input. This is called the extraction operator. The function prototype of the operator function will be as follows:

```
istream &operator>>(int &var);
```

All the overloaded functions of >> return a reference to the `cin` object. The argument to the function is taken as a reference since it has to store the input value and reflect it back in the calling function. Also the function returns a reference to the `cin` object so that more than one variable can be accepted with a single statement such as,

```
cin>>name>>age>>salary;
```

While reading input the entire overloaded operator functions skip over white spaces - blanks, newlines and tabs - until a non-white space character is encountered. For numbers, it reads everything after initial white space characters, until a non-numeric character.

A.6 Other istream functions

The `istream` class also defines the `get()` and `getline()` functions for input. Some C++ implementations do not support `get()` and `getline()`. The `get()` function has the following prototype:

```
istream &get(char &ch);
int get();
```

The first function stores the character entered in `ch` and returns a reference to the `istream` object. The second form returns back the character entered. Both the functions fetch the very next character entered, even if it is a white space.

The `getline()` and overloaded `get()` function accept strings. Their function prototypes are:

```
istream &get(char* s, int size, char ch = '\n');
istream &getline(char* s, int size, char ch ='\n');
```

The first argument is the address of the location from where the characters are read. The second argument is the number of maximum characters to be read (plus one for the null character). The third argument is the terminating character. If that is omitted, the functions read upto the maximum characters specified or until a newline character, whichever comes first. For example,

```
char string[100];
cin.get(string,50);
```

Durr-e-Danish Library

Scanned by CamScanner

The function reads upto 49 characters or until a newline character, whichever is first. The difference between `get()` and `getline()` is that `get()` leaves the newline character in the input stream, whereas `getline()` extracts and discards it from the stream.

There are other functions in `istream` to facilitate input, like `read()`, `peek()`, `gcount()`,`putback()`, and `ignore()`. The prototypes of these functions are:

```
istream &read(char *line, int size);
```

This function reads size number of characters from the standard input and stores them in line. It does not append a null character to the accepted input.

```
int peek();
```

This function returns the next character from the standard input, but leaves it intact in the buffer. It is a sort of look-ahead function.

```
int gcount();
```

This function returns the number of characters read from the standard input, by the last input function from `istream`. The last input function must be `read()`, `get()` and `getline()` only and not the overloaded extraction insertion operator.

```
istream &putback(char ch);
```

This function inserts the characters given by `ch` in the input stream.

```
istream ignore(int len =1, int ch = EOF);
```

This function reads and discards the next `len` number of characters or until the first `ch` character, form the standard input. It discards the terminating character as well.

Durr-e-Danish Library

Scanned by CamScanner

This page has been intentionally left blank

Durr-e-Danish Library

Scanned by CamScanner

A preprocessor directive is an instruction to the compiler. The part of the compiler called the preprocessor deals with the directives in the source code before the compilation process. The output from the preprocessor becomes the input to the compiler and is also called as the extended source code.

All the preprocessor directives are preceded by the # symbol. The commonly used preprocessor directives are:

```
#include
#define
#undef
#ifdef, #elif, #else and #endif
#error
```

B.1 #include

The #include mechanism is a text manipulation facility for combining program segments into a single file for compilation. The directive,

```
#include "my_file"
```

replaces the line in which the #include appears in the program with the contents of the files my_file. To include files from the standard include directory we use the angle brackets < and >. The quotes indicate that the file to be included is in the current directory. For example,

```
#include<iostream.h>          //from standard include directory
#include "header.h"           //from current directory
```

B.2 The #define directive

The #define directive is used for defining simple macros that can be used to specify symbolic constants. The general form for a simple macro definition is,

```
#define macro-name sequence-of-tokens
```

The directive associates the sequence-of-tokens with the macro-name. Whenever the preprocessor finds the macro-name in a program, it is replaced by the sequence of tokens it represents. Macro names are not recognised within comments or string constants. The following is an example, which defines a value 3.14285127 and gives it a name PI.

```
#define PI 3.1485127

void main()
{
  cout<<PI;
}
```

This program is equivalent to:

Durr-e-Danish Library
Scanned by CamScanner

```
void main()
{
 cout<<3.14285127;
}
```

The #define macros work strictly on the basis of substitution of the text string for the macro names.

➢ **Macros with parameters**

Macros can also be defined with arguments. The general form used is:

```
#define macro-name(arg1,arg2,. . .) sequence of tokens
```

Macros with parameters are primarily used to define functions that expand into inline code. For example:

```
#define abc(x,y) code
```

where x is argument1 and y is argument2. Thus, when abc() is used, two argument strings must be used. When abc() is expanded the arguments passed will replace x and y. The preprocessor first replaces the formal parameters in the body of the macro by the actual arguments, and then the resulting macro body is substituted for the macro call. Some examples of macros with parameters are:

```
#define ABS(N)          ((N)>=0? (N): -(N))
#define ADD(X,Y)        ((X)+(Y))
#define RATIO(X,Y)      ((X)/(Y))
```

Macro names cannot be overloaded. Macros operate purely by textual substitution of tokens. The compiler sees only the expanded form of a macro. Any error in a macro will be reported by the compiler only when the macro is expanded and not when it is defined.

A macro definition, independent of the block structure, lasts until the end of the program unit. If you refer to global variable names in a macro the scope resolution operator must be used.

B.3 The #undef directive

This directive un-defines a previously defined macro. The directive is given in the form

```
#undef identifier
```

where identifier is a macro name. If the identifier is not currently defined as a macro name this directive is ignored. The following program will give a syntax error, since PI is undefined.

```
#define PI 3.1485127

void main()
{
 #undef PI
```

Durr-e-Danish Library

Scanned by CamScanner

```
  cout<<PI;
}
```

A macro can be redefined, provided it has been previously undefined by the `#undef` directive.

B.5 Conditional compilation

Conditional compilation allows selective inclusion of lines of source text on the basis of a computed condition. The directives `#ifdef`, `#elif`, `#else` and `#endif` are used for conditional compilation. They can be used in the following ways:

```
➢ #ifdef condition
     statements;
  #endif
```

```
➢ #ifdef condition
     statements;
  #else
     statements;
  #endif
```

```
➢ #ifdef condition1
     statements
  #elif condition2
     statements;
  #else
     statements;
  #endif
```

A directive of the form

```
#ifdef condition
```

checks whether the condition evaluates to true or false. Like `#ifdef` there is `#ifndef` which negates the condition.

The `#else` directive indicates alternatives when the previous `#if`, `#ifdef`, or `#ifndef` test fails. The `#endif` directive ends the conditional text. There must be a matching `#endif` for every `#if`, `#ifdef`, or `#ifndef` directive. The `#elif` directive allows a more convenient form by combining `#else` and `#if` directives and avoids multiple `#endif` directives.

Each directive's condition is evaluated in order. If the condition evaluates to false, the associated group of lines is skipped. Only the first group of lines whose control condition evaluates to true is included. If none of the conditions evaluates to true, the group of lines associated with `#else` is included. If there is no `#else`, all the lines until `#endif` are skipped. For example,

```
#ifndef MAX
      #define MAX 6
```


Durr-e-Danish Library
Scanned by CamScanner

```
#else
      #undef MAX
      #define MAX 10
#endif
```

These directives control which portions of the code will be passed to the compiler. For example, these directives can also be used for compiling a program written for two different machines. On DOS-based machines, the program can make use of certain MS-DOS features. Otherwise, use some other code depending on the machine.

B.6 The #error directive

This directive allows you to put diagnostic messages into the preprocessor statements. Sometimes, working with conditional compilation, error messages need to be displayed. Sometimes a program has to be compiled with specific options. In these cases this directive is very useful. The following example illustrates the usage of this directive.

```
#ifndef ADD
#error "operator not specified"
#endif

void main()
{
  ...
}
```

If it is found during compilation of the program that ADD is not defined, the preprocessor prints the given error message and stops compilation.

Durr-e-Danish Library
Scanned by CamScanner

C.1 Errors

While writing programs it is inevitable that there will be some errors - either typographical mistakes or errors which the system generates because of wrong syntax. The errors can be detected by the compiler or by the linker or they can appear at run-time.

➢ Compiler errors

In a program that you have written, if you have forgotten to type a semicolon at the end of a statement the program will not compile. You will see that the Compiling window, will show Errors instead of the Success message. The Message window that appears at the bottom of the screen will display the error.

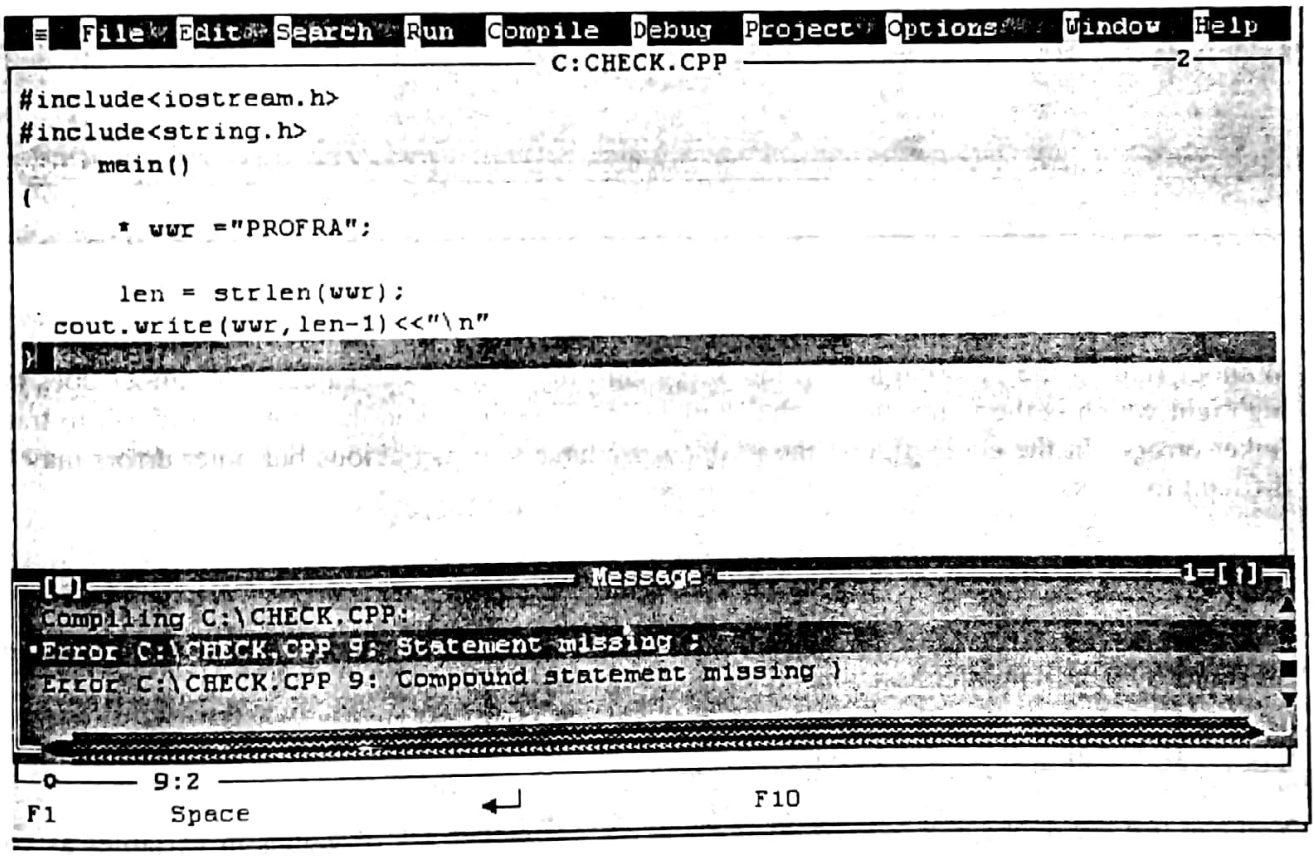


Figure 1

In the Edit window, the line with the closing brace is highlighted. This indicates the place where the compiler found the error. Press F6 to move from the Message window to the Edit window. Once you correct the mistake the program will compile correctly.

➢ Linker errors

During the linking process also errors can occur. If you have misspelled the word `main` as `maid` the program will compile correctly. However, the linking process will result in an error. The Message window will show a message as shown in Figure 2.

Durr-e-Danish Library
Scanned by CamScanner

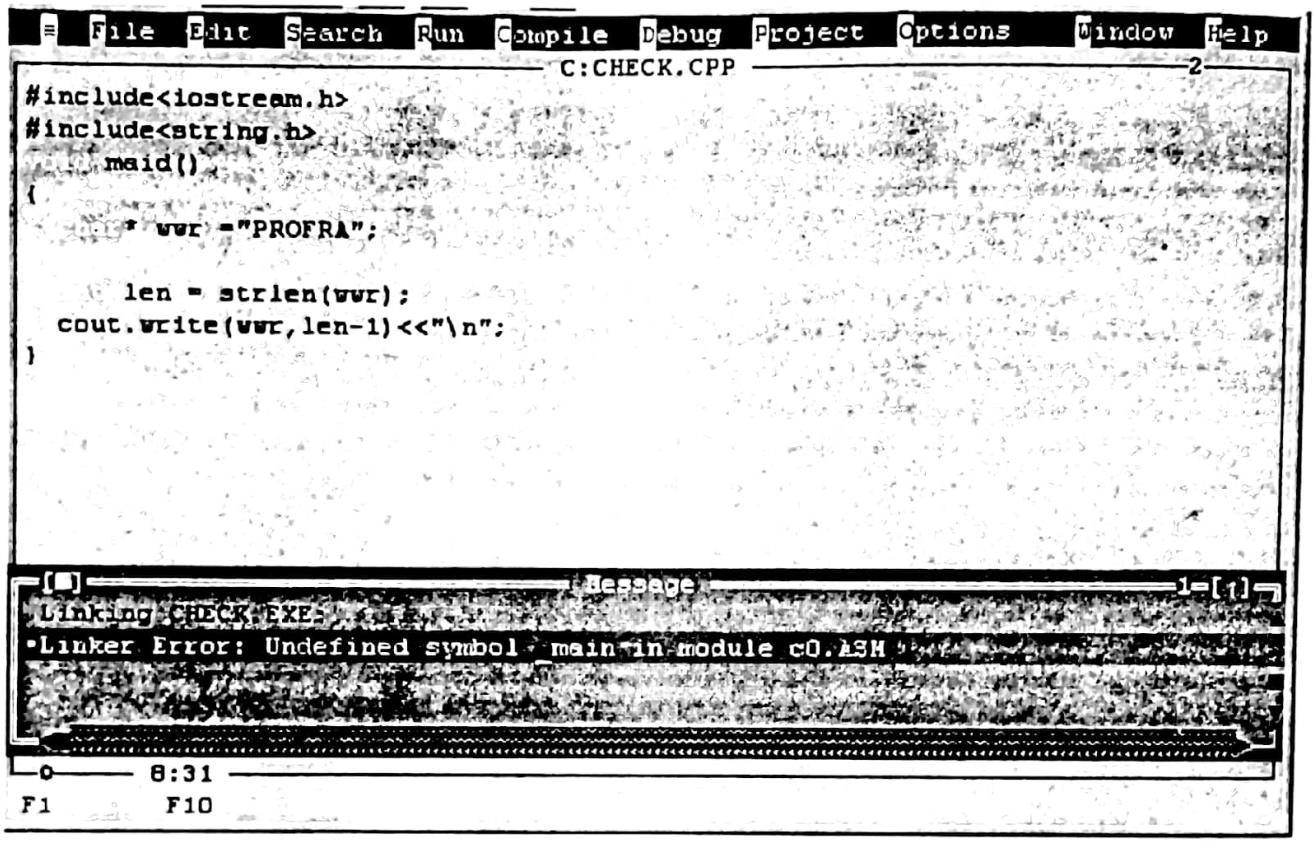


Figure 2

Without finding the function `main`, the executable file cannot be created. The linker does not highlight which is the faulty line in the source file. Therefore, you may find it difficult to track linker errors. In the above figure, the problem we have seen is obvious but other errors may be difficult to track.

- **Run-time errors**

There are a few errors that do not emerge till the program executes. These errors include divisions by 0, null pointer assignment, and a few others.

C.2 Debugging Features of Borland C++

You can use the debugging features of Borland C++ to keep track of values in variables as you are running a program. This process is called single-stepping.

Open your program in the Edit window and press F8. If necessary, the program will be recompiled. The first executable line in your program (usually `void main()`)will be highlighted. This indicates that the highlighted line is about to be executed. As you press F8 repeatedly, the highlight moves through the program till the end of the program. In a for loop or while loop, you will see that the highlighted line will keep moving through the loop till it is completed.

Durr-e-Danish Library

Scanned by CamScanner

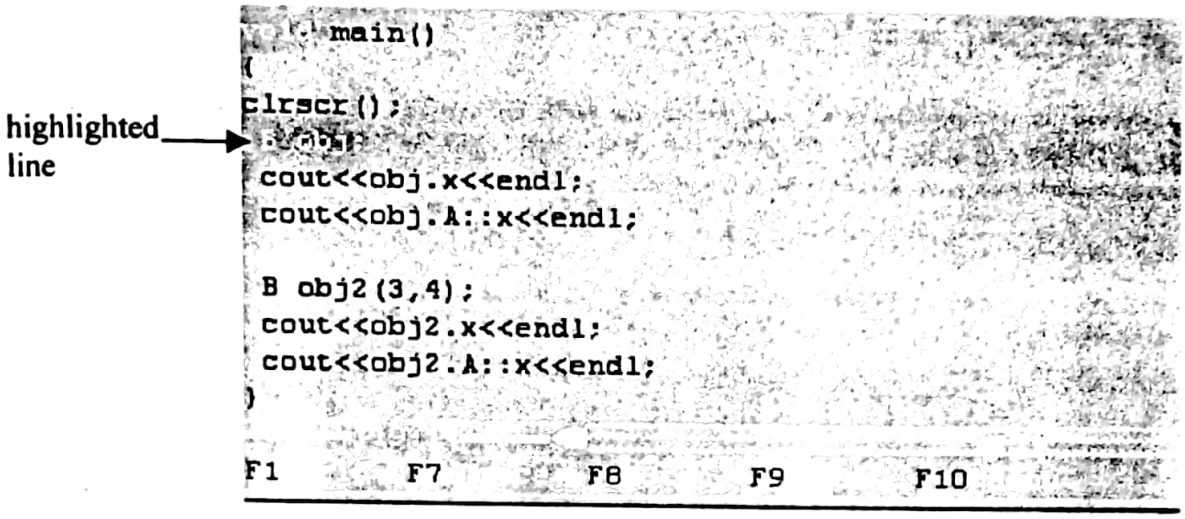


Figure 3

- **The Watch window**

It is useful to see the values of variables as the program is running, so that you can check that your program is generating the output you want. To do this you can open a Watch window by selecting Window | Watch from the main menu. Select Watches from the Debug menu and from the submenu select Add watch.

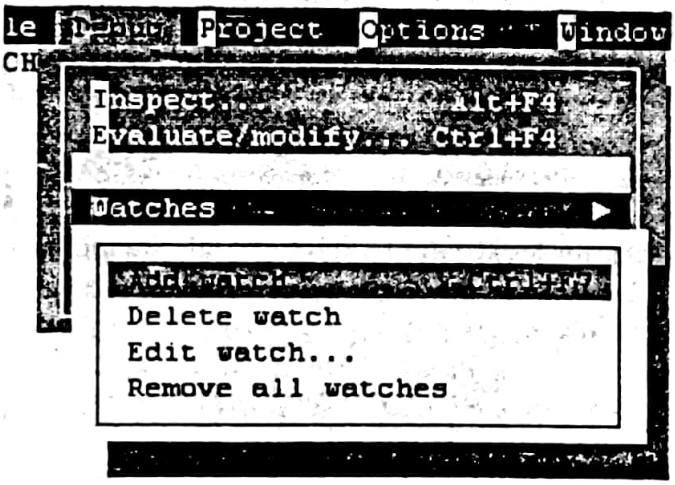


Figure 4

In the dialog box that appears, you can enter the name of the variable to be watched.

Durr-e-Danish Library
Scanned by CamScanner

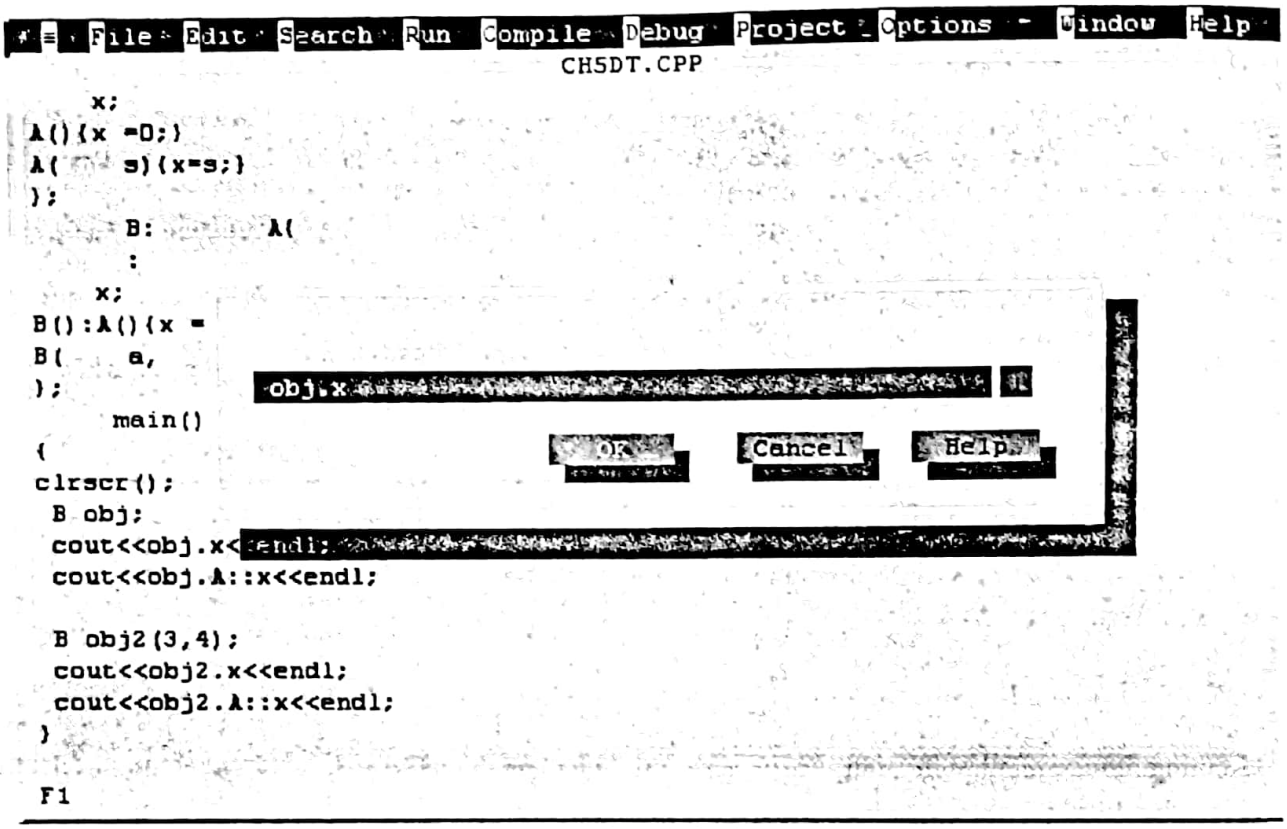


Figure 5

The Watch window that will appear at the bottom of the screen will display the value of the variable. As you step through the program the variable in the Watch window will reflect the changes in its value. You can add as many variables as you like to the Watch window. The values for all the variables will be displayed.

If you find that the variable you are monitoring is not showing the values you expect it to, you should check your program for logical errors. Using single-stepping and the Watch window will usually help you to locate the source of the problem. Using these techniques is especially useful in tracking the values of pointers.

Durr-e-Danish Library

Scanned by CamScanner

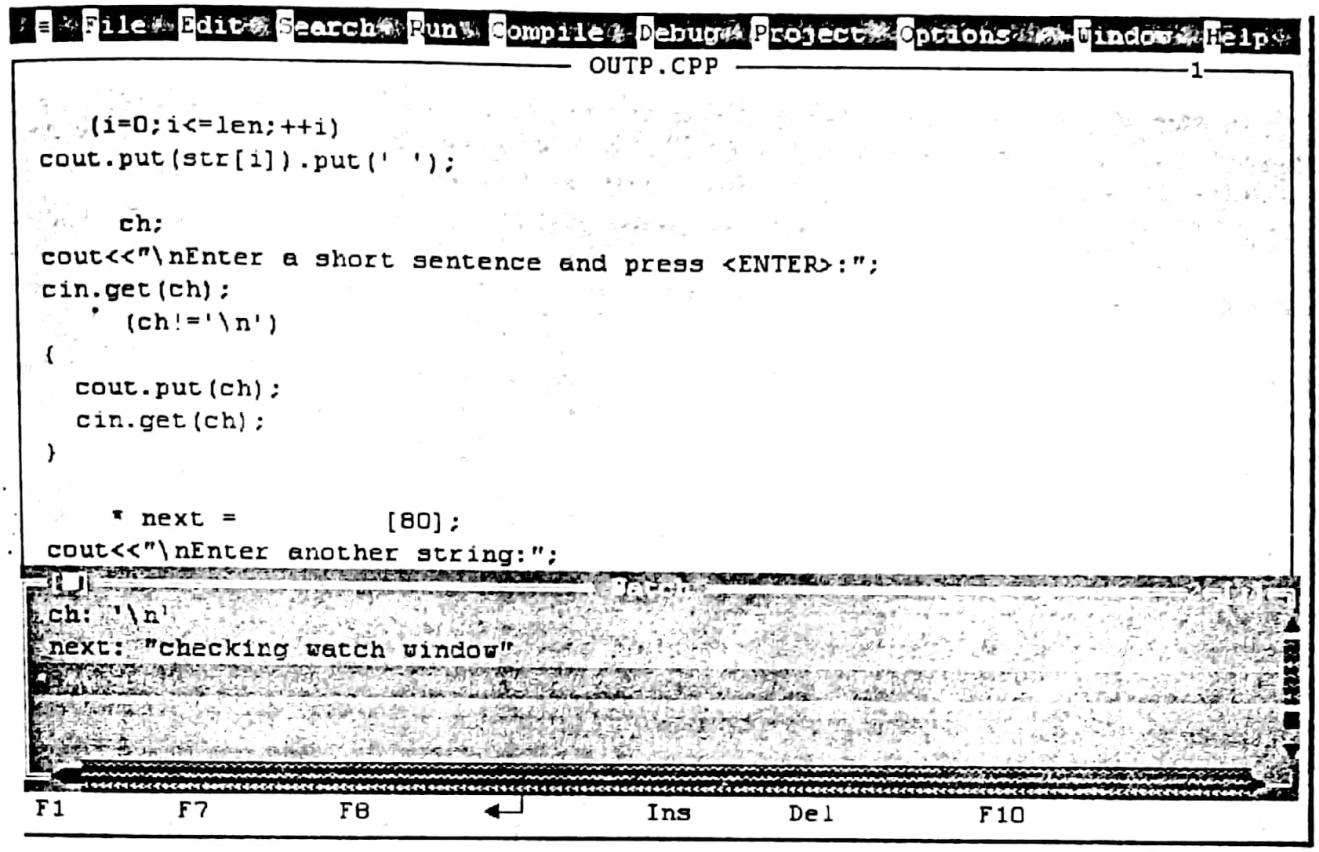


Figure 6

Durr-e-Danish Library
Scanned by CamScanner

Operator

The dot operator is called the class member operator. A member function is associated with a specific object with the dot operator (the period).

<<Operator

is called the insertion operator. It is overloaded by the cout object for doing output.

>>Operator

is called the extraction operator. It is overloaded by the cin object for input

Abstract class

An abstract class is a class that can be used only as a base class of some other class. No objects of the class can be created. Contains one or more pure virtual functions.

Access specifier

An access specifier specifies the access rules for members following it until the end of the class or until another access-specifier is encountered. The three access specifiers are public, private and protected.

class

A class is a grouping of objects that have the same properties, common behaviour and common relationships.

cin

An object of istream class. Represents the standard input stream, which is the keyboard.

const keyword

The keyword const can be added to the declaration of an object to make that object a constant rather than a variable.

Constructor

A constructor is a special member function for automatic initialisation of an object.

Container class

When an object of a class is declared as a member of another class. it is called *a container class*.

Durr-e-Danish Library

Scanned by CamScanner

Conversion functions

Conversion functions are member functions used to convert objects to or from basic data types and for conversions between objects of different classes.

Copy constructor

A copy constructor for a class X is a constructor that can be called to copy an object of class X.

cout

An object of the ostream class. Represents the standard output stream, which is the video display.

Data Abstraction

Data abstraction is the process of identifying properties and methods related to a particular entity as relevant to the application.

Default Constructor

A default constructor is a constructor that does not have any arguments.

delete operator

The delete operator frees the memory that was previously allocated using the new operator at run time.

Destructor

A destructor is a member function that is called automatically when an object is destroyed.

Direct Base Class

A class is called a direct base if it is mentioned in the base-list of the derived class.

Dynamic binding

Dynamic binding means that the address of the code in a member function invocation is determined at the last possible moment, based on the dynamic type of the object at run time.

Dynamic memory management

Most C++ compiler libraries provide functions that allow you to allocate and free data space while the program is executing. This kind of management is called dynamic memory management.

Encapsulation

Encapsulation is the process that allows selective hiding of properties and methods in a class or it is the process that allows selective exposing of properties and methods in a class.

Durr-e-Danish Library

Scanned by CamScanner

Friend Function

A friend function is a non-member function that is allowed access to the private part of a class.

Function overloading

Function overloading is used to define a set of functions that are given the same name and perform basically the same operations, but use different argument lists.

Function prototype

The declaration that describes the function's name, its return value and the number and types of its parameters.

Indirect Base Class

A class is called an indirect base if it is the base class of one of the classes mentioned in the base-list of the derived class.

Inheritance

Inheritance is the property that allows the reuse of an existing class to build a new class.

inline function

An inline function is written just like a normal function in the source file but compiles into inline code instead of into a function.

Member Functions

A function declared as a member of a class is called a member function.

Method

An action required of an object or entity when represented in a class is called a method.

Multiple Inheritance

Multiple inheritance is the process of creating a new class from more than one base class.

new operator

The new operator is used to allocate memory space and return a pointer to that memory.

Object

An object is a concept or thing with defined boundaries that is relevant to the problem we are dealing with.

Durr-e-Danish Library

Scanned by CamScanner

operator function

An operator function is a special member function that overloads an operator. The operator that has to be overloaded follows the keyword operator.

Operator overloading

The ability to associate an existing operator with a member function and use it with objects of its class as its operands is called operator overloading.

Polymorphism

Polymorphism indicates the possibility that an entity can take many forms. It is the process of defining a number of objects of different classes in a group and using different function calls to carry out the operations of the objects.

Pointer

A pointer is the address of a byte in the computer's memory.

Property

A characteristic required of an object or entity when represented in a class is called a property.

Pure virtual function

A pure virtual function is a virtual function with the pure specifier in the function declaration in the class declaration. It has only a function declaration.

Reference

A reference provides an alias or an alternate name for an object.

Single Inheritance

Single inheritance is the process of creating new classes from an existing base class.

Static data member

If a data item in a class is defined as static, then only one such item is created for the entire class, no matter how many objects there are.

Structure

Structure organises different data items so that they can be referenced as a single unit.

Superclass

The superclass is the class from which another class inherits its behaviour.

Durr-e-Danish Library

Scanned by CamScanner

Subclass

The class that inherits the properties and methods of another class is called the subclass.

this pointer

Whenever a member function is called, the compiler assigns the address of the object which invoked the function, to the `this` pointer. The keyword `this` gives the address of the object, which was used to invoke the member function.

Type casting

In C++ the term type casting applies to data conversions that are specified by the programmer.

Virtual function

A virtual function allows derived classes to replace the implementation provided by the base class.

Virtual base class

Any base class that is declared using the keyword virtual is called a virtual base class.

Durr-e-Danish Library

Scanned by CamScanner

APTECH

COMPUTER EDUCATION

we *change* lives

Scanned by CamScanner